Memory Caches: Structure

CS 3220 / CS 5220 Lecture 2-B

Jason Hibbeler

University of Vermont

Spring 2024

Overview

Memory stalls

Cache structures and associativity

Cache-write options

Optimizing cache performance

Memory Cache

A smaller, high-speed memory that sits between the CPU and main memory

Cache access is fast

- cache access time is one clock cycle
- memory access time is 100 clock cycles

Goal:

- keep data that will be used again soon in the cache

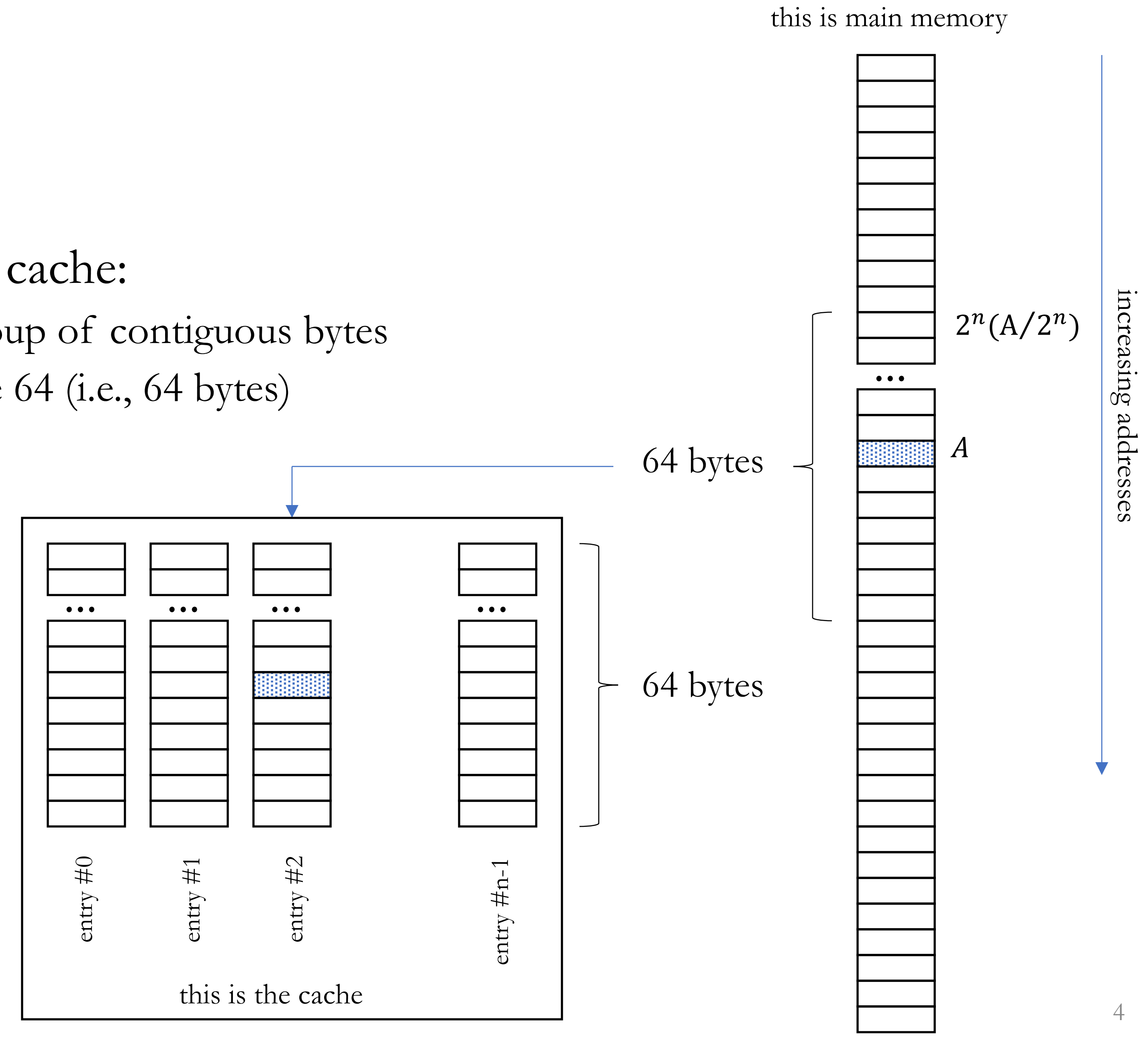
Cache Blocks

Bringing data from memory to the cache:

- we bring in a cache block—a whole group of contiguous bytes
- a typical size cache block will be of size 64 (i.e., 64 bytes)

A cache miss to the address A results in 64 bytes being brought into the cache

If each block is of size 2^n, then the block we bring in starts at address $2^n(A/2^n)$: we'll call the value $A/2^n$ the *block address*



Cache Blocks

In other words: if the size of each cache block is 2^n

Then for a memory access to address A that causes a cache miss

- round the address A down to the nearest multiple of 2^n: this is $2^n(A/2^n)$, using integer division
- and bring 2^n bytes into the cache
- this will include the word at address A
- as well as 2^n bytes near A

Example: if $A = 1000$ and $n = 6$: $64(1000/64) = 64\times15 = 960$

Analogy: Library

Suppose I'm writing a research report

- I have my own private study cubicle in the library
- my cubicle has a small bookshelf on it
- I'll keep the books and journals that I'm currently using on the bookshelf in my study cubicle
- the library itself has a vast collection of books and journals
- it takes a long time to fetch a particular item from the general collection
- but it's quick for me to access information in a book or journal that's on my bookshelf

Now, suppose I need issue #12 of the journal *Nature*

- I'll fetch issue #12 and put it on my bookshelf
- but I'll also fetch issues #10, #11, #13, #14, and #15, because I might need them soon
- and the next time I need to reference issue #12 (or issue #10, etc.), it will be much faster

Cache Architecture

The motivation for a cache is clear

- put data that will be needed soon nearby

However, there are many details to consider

Questions Concerning Cache Architecture

1. Where can a block be placed in the cache?
2. How is a block found if it is already in the cache?
3. Which block should be replaced on a cache miss?
4. What happens on a write?

Question 1: Where Can a Block Be Placed?

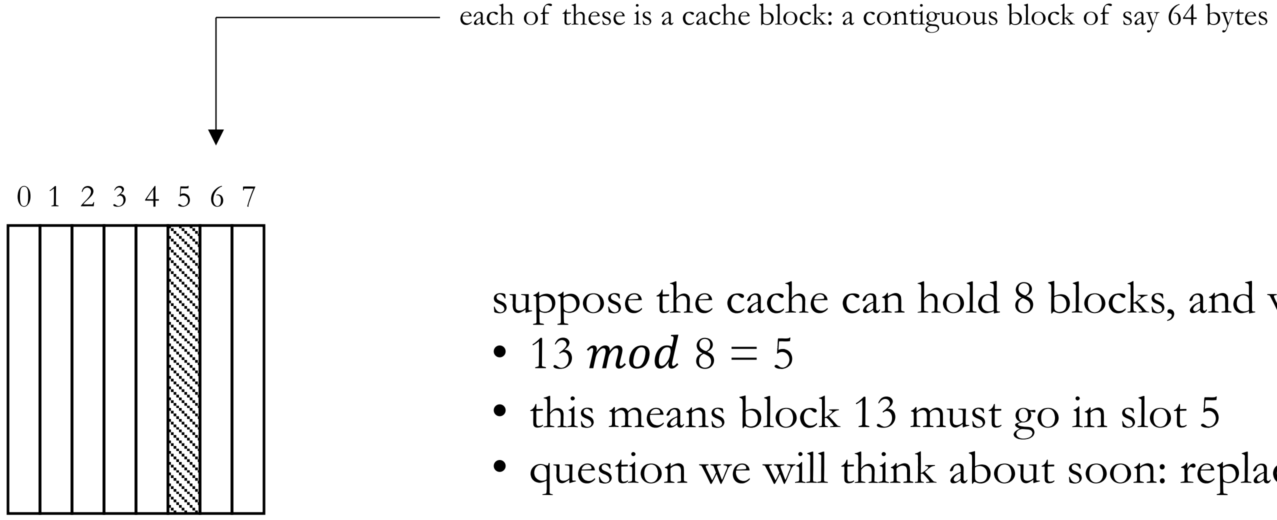
There are two basic cache structures

1. direct mapped
2. associative

Structure #1: Direct-Mapped Cache

In a *direct-mapped* cache, each block has only one place (one “slot”) in which it can appear in the cache

- and the location in the cache is typically given by (block address) mod (#blocks in cache)


each of these is a cache block: a contiguous block of say 64 bytes
0 1 2 3 4 5 6 7

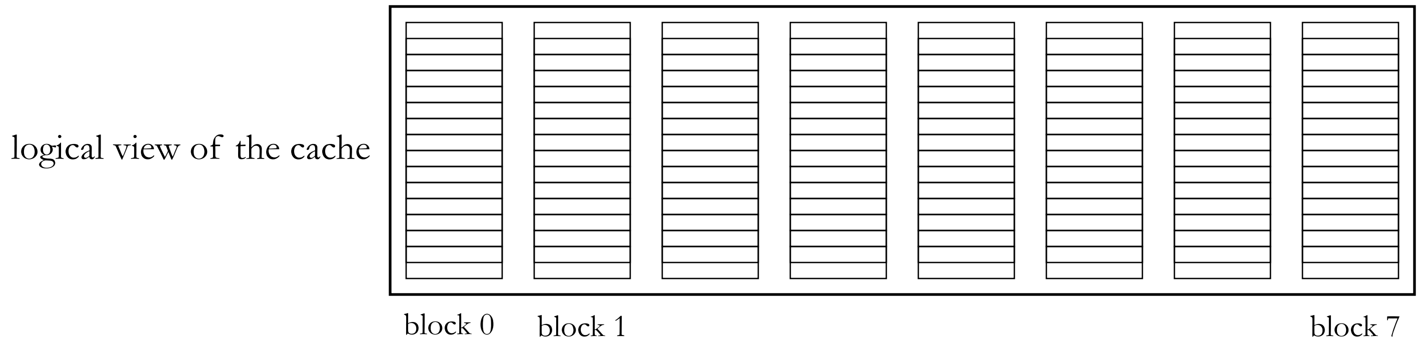
suppose the cache can hold 8 blocks, and we fetch block 13 from memory

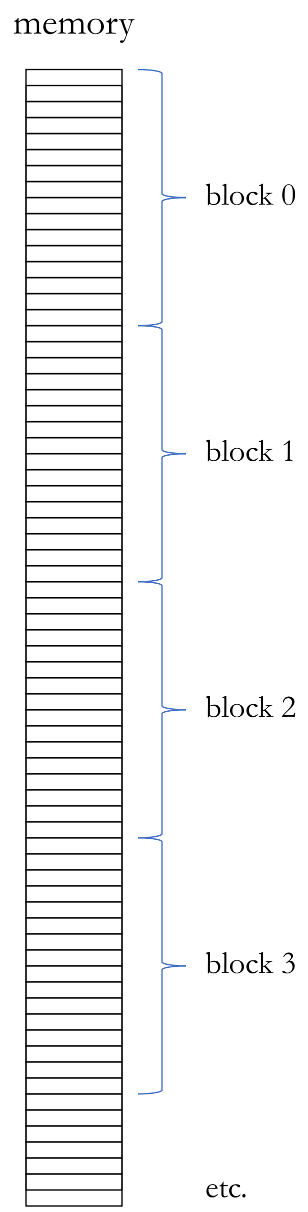
- 13 mod 8 = 5
- this means block 13 must go in slot 5
- question we will think about soon: replacement strategy (for collisions)

Reading a Block

In an 8-block direct-mapped cache, where each block is of size $2^4 = 16$

- the block address of A is $A/2^4$ (integer division)
- call this B_A
- so in response to a cache miss for a reference to address A, the memory subsystem would read the 16 bytes starting at $16 \times B_A = 16(A/16)$ and put those 16 bytes (this is the block containing A) into the cache
- the actual location of B_A in a direct-mapped cache is given by $B_A\ mod\ m$, where m is the number of blocks in the cache; so here, it's $B_A\ mod\ 8$
- the actual location of A in the cache block is $A\ mod\ 16$

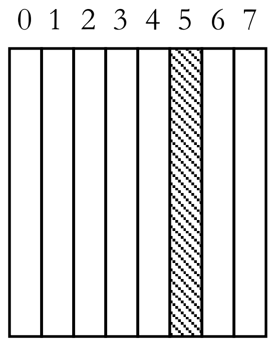




Direct-Mapped Cache

In a direct-mapped cache, each block has only one place (one "slot") in which it can appear in the cache

- and the location in the cache is given by (block address) mod (#blocks in cache)



Suppose this cache has 8 blocks
And suppose we fetch block 13 from memory

- $13\ mod\ 8 = 5$
- this means that block 13 must go in slot 5, and whatever was in slot 5 is lost

Now suppose we fetch block 21 from memory

- $21\ mod\ 8 = 5$
- this means that block 21 must also go in slot 5, and whatever was in slot 5 is lost

Suppose we access block 13 and then block 21 and then block 13 etc.

- each access will cause a cache miss and block replacement
- this is expensive (in terms of time) and inefficient!

Library Analogy, Again

The bookshelf in my cubicle is small (bookshelf in my cubicle: the cache)

- only a few things will fit on it

Each item that I fetch from the general collection (main memory) has a specific place where it will need to be on my bookshelf

- issues #10-15 of *Nature* will go in position #5
- so if I fetch these from the collection, I'll put them on my shelf in this position
- and whatever was there (if there was anything there) will have to go back to the stacks

Suppose that in my mapping scheme, issues #48-53 of *Martha Stewart Living* also map to position #5

- then both of these groups of journals cannot be on my bookshelf at the same time

Direct-Mapped Cache: Cache Lookup

In a direct-mapped cache, more than one block from memory will map to the same cache block

- for cache block b in the cache, every block B such that $B \; mod \; m = b$ will map to b

So, how can we tell whether an address that lies in block B is actually in the cache?

- for example, blocks 21 and 13 both map to block #5 in the previous example

Somehow, the cache needs to keep track of which memory block is actually in cache block #5 in the cache

Direct-Mapped Cache: Cache Lookup

How can we tell whether an address that lies in block B is actually in the cache?

Solution: tags

- put additional data in the cache showing which memory block is actually present in a particular slot
- this data is called a *tag*
- it's formed from the actual memory address of the data that resides in that slot in the cache

So now, a cache lookup has two steps

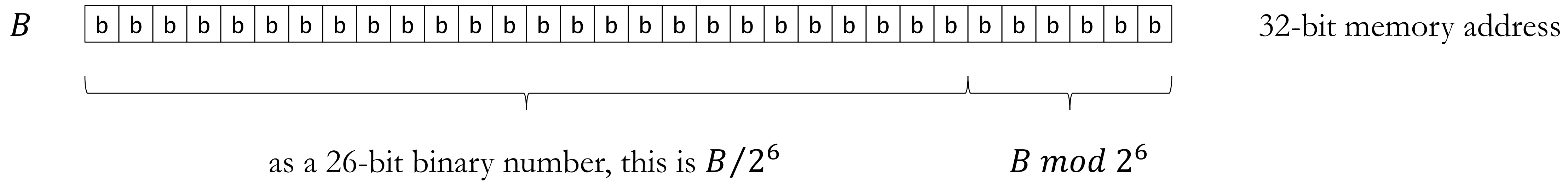
- go to the place in the cache given by $B \ mod \ m$: this is where the block would be in the cache if it actually is in the cache
- check the tag to see whether B is actually in the cache at that slot

Bit Operations

The memory-block-to-cache-block mapping, through integer division, is actually a bit operation

- if B is a binary number having r bits, then the calculation $B/2^m$ is equivalent to taking the most significant $r - m$ bits of B
- similarly, $B \ mod \ 2^m$ is equivalent to taking the least significant m bits of B

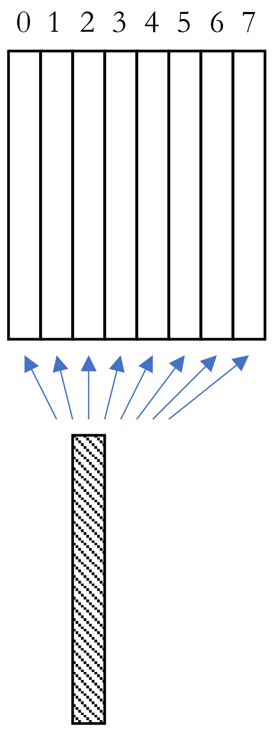
Suppose $r = 32$ and $m = 6$


B
b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b b
32-bit memory address
as a 26-bit binary number, this is $B/2^6$
$B \ mod \ 2^6$

Structure #2: Associative Cache

In a *fully associative* cache, a block can go into any slot

- this will prevent the potential constant cache misses we saw in the direct-mapped cache



block 13 from memory can go into any slot

- the actual decision of where a block should go is complex
- question to think about: replacement strategy

From before: suppose this cache has 8 blocks
And suppose we fetch block 13 from memory

- block 13 can go into one slot

Now suppose we fetch block 21 from memory

- block 21 can go into a different slot

And we can repeatedly access block 13 - block 21 - block 13 - block 21

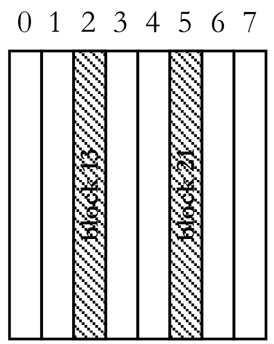
There will not be any collisions

- both blocks can fit in the cache!

Associative Cache

In a fully associative cache, a block can go into any slot

- this will prevent the potential constant cache misses we saw in the direct-mapped cache



From before: suppose this cache has 8 blocks
And suppose we fetch block 13 from memory
- block 13 can go into one slot

Now suppose we fetch block 21 from memory
- block 21 can go into a different slot

And we can repeatedly access block 13 - block 21 - block 13 - block 21

There will not be any collisions
- both blocks can fit in the cache!

block 13 and block 21 can both fit in the cache
- the tag values in the cache say which actual memory block is in each cache block

Analogy

In my library cubicle, I'll let both groups of journals be placed anywhere on my shelf

Associative Caches

This is good, but there's a drawback

- it takes time and power to look at tags

In the worst case, the cache hardware would have to look all *m* tags in a fully-associate cache of size *m*

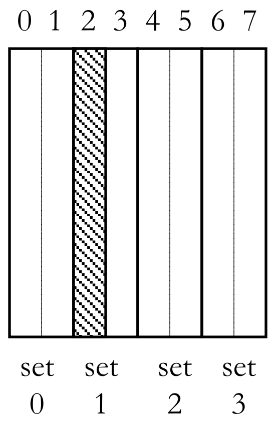
Idea for a compromise: instead of letting a block occupy any slot in the cache

- let a block occupy a reduced set of blocks
- partition the cache into sets

Set-Associative Cache

Set-associative cache:

- a given block can be placed in a restricted set of slots in the cache
- essentially, a combination of direct-mapped and associative approaches



Example: 2-way set-associative cache, with four sets

- block 13 from memory can go into either slot of set 1
- set 1, because 13 % 4 = 1 (4, because there are four sets)
- the actual decision of which slot in the set block 13 should occupy is more complex
- question to think about: replacement strategy

But now, if we have a sequence of memory accesses to block 13 and block 5 over and over again, then:

- both block 13 and block 5 can sit in the cache
- they'll be in the same set (13 % 4 = 1, and 5 % 4 = 1*)*
- but in different slots in that set (set #1)

Cache Structure

A direct-mapped cache having m blocks is really just one-way set associative

- in other words, there are m sets, and the number of blocks in each set is 1

A fully associative cache with m blocks is really just m-way set associative

- a block can go anywhere—in any of the m blocks; so the set size is m
- but there is a single set

Cache Misses

We can categorize a cache miss as one of three different types:

(1) Compulsory miss: when the cache is not full and the destination block for a memory miss is empty; this happens the first time a program requests a particular block

(2) Conflict miss: when the cache is not full but the destination block for a memory miss is currently occupied by a different block

- the key thing is that with higher associativity, this miss would be avoided

(3) Capacity miss: when the cache is already full (including the destination block for a memory miss)

- the only way to avoid this is with a larger cache

Question 2: How is a Block Found in the Cache?

Each slot in the cache has an identifier called a tag

- the tag shows what range of addresses (i.e., which block) is actually located in that cache block

Memory address		
Block address		Block offset
Tag	Index	Block offset
$r - m - n$ bits, for an r-bit address	m bits, for 2^m sets	n bits, for block size 2^n

The components of a memory address

Components of an address:

- block offset: where am I in the the cache block?
- index: which set am I in?
- tag: this is what we check to find out which actual cache block in the target set contains this address

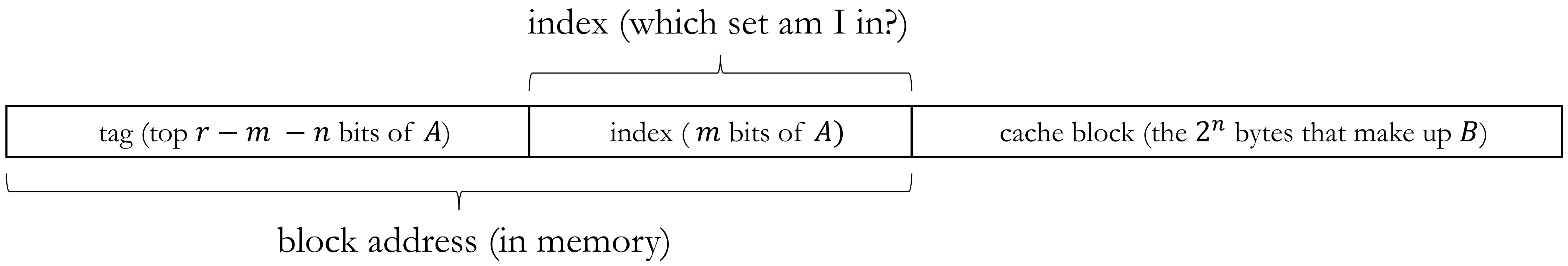
Tags for Direct Mapping

More than a single memory block B will map to each cache block

- so the question is: for address A, having block address B_A, how do we know that the cache block in position $B_A\ mod\ m$ actually contains B_A?
- keep the most significant $r - m - n$ bits of the address in the cache itself next to the block B: this is the tag

So for a cache having 2^m blocks and block size = 2^n

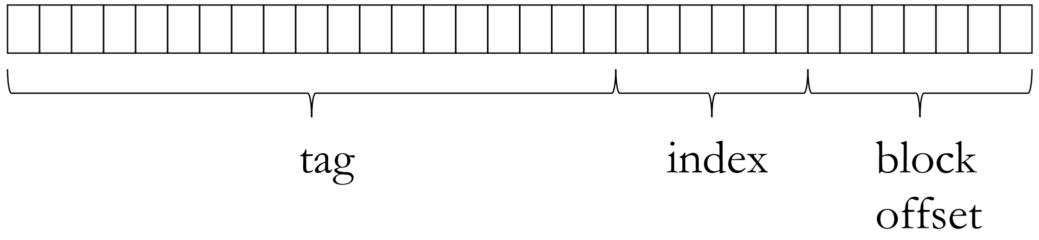
- each cache entry would look like this:


index (which set am I in?)
tag (top $r - m - n$ bits of A)
index (m bits of A)
cache block (the 2^n bytes that make up B)
block address (in memory)

Example #1

Suppose we have the following:

- 32-bit memory addresses
- a 16K cache (16K = 2^{14})
- 64-byte cache blocks ($64 = 2^6$)
- four-way set associativity



Then

- there are $2^{14}/2^6 = 256$ cache blocks
- but only $256/4 = 64$ cache sets (because each set contains four blocks: four-way set associativity)
- we need 6 bits for set selection (the index) and 6 bits for byte selection in a block
- so the tag is $32 - 6 - 6 = 20$ bits in length
- the index tells us which set we're in
- but we need to check the tag to see which actual block in that set is the correct one

Example #1

With the memory and cache from the previous slide:

- suppose that the address is 1984832562

A = 1984832564	0 1 1 1 0 1 1 0 0 1 0 0 1 1 1 0 0 0 1 0 0 1 0 0 0 0 1 1 0 1 0 0	
tag = 484578	0 1 1 1 0 1 1 0 0 1 0 0 1 1 1 0 0 0 1 0	
index = 16	0 1 0 0 0 0	m bits, for 2^m sets
block offset = 52	1 1 0 1 0 0	n bits, for block size 2^n bytes

We can't determine the block index from the address—it's a function of the replacement policy of the cache

- the index is the set in which the cache block is located
- the block index is the position of the block in the target set

Example #2: AMD

AMD Opteron: 40-bit physical address

- 64 KB two-way set-associative cache (64 KB = 2^{16})
- 64-byte cache blocks ($64 = 2^6$)
- this means that there are $2^{16}/2^6 = 2^{10}$ different cache blocks
- two-way set associativity means that there are $2^9 = 512$ different sets
- so we need 9 bits to determine the set and 6 bits to determine the offset inside of a cache block
- this leaves $40 - 9 - 6 = 25$ bits as the size of the tag

Each cache block thus has a 25-bit tag that the cache hardware checks to see whether the data for a particular address is contained within the target cache block

Example #3

Suppose we have:

- 16-bit addresses
- 16-byte cache blocks
- a 256-byte cache

This means there are 16 blocks (256 bytes / 16 bytes per block)

We'll look at two different caches

- direct mapped: this means there are 16 sets (each set has one block)
- two-way set associative: this means there are eight sets (each set has two blocks)

Example #3, with a Direct Mapped Cache

16 sets, and each set has one block

- lowest four bits: block offset (position of the byte in the block); four bits because each block has 16 bytes
- bits 4-7: the index (which set we're in); four bits, since there are 16 sets
- top 8 bits: the tag—uniquely identifies this memory block in the cache; this is everything else in the address
- since this is a direct-mapped cache, each set has only one block, and block index is always zero

address	binary	index	tag	block index	block offset	memory range for this block
5000	00010011 1000 1000	8	19	0	8	4992 - 5007
5004	00010011 1000 1100	8	19	0	12	4992 - 5007
5008	00010011 1001 0000	9	19	0	0	5008 - 5023
144	00000000 1001 0000	9	0	0	0	144 - 159

Example #3, with Two-Way Set Associative

Eight sets, and each set has two blocks

- lowest four bits: block offset (position of the byte in the block); four bits, since each block has 16 bytes
- bits 4-6: the index (which set we're in); three bits, since there are eight sets
- top 9 bits: the tag—uniquely identifies this memory block in the cache
- block index: determines which block in a set we use

address	binary	index	tag	block index	block offset	memory range for this block	comment
5000	000100111 000 1000	0	39	0	8	4992 - 5007	assume first in this set
5004	000100111 000 1100	0	39	0	12	4992 - 5007	
5008	000100111 001 0000	1	39	0	0	5008 - 5023	assume first in this set
144	000000001 001 0000	1	1	1	0	144 - 159	second in this set

Intermission

Summary so far

Suppose the cache has block size $b = 2^n$

- we can consider the memory to be divided into blocks of size b
- each block starts at an even multiple of b
- when a read of address A inside of block B is made, the whole block is brought into the cache
- there will be only a single set in the cache where a particular block can reside
- the set could have either one slot (in a direct-mapped cache) or k slots (in an k-way set-associative cache)

Summary So Far

The specific set in the cache having block size 2^n where the block containing address A will reside is a given by a function $f(A)$

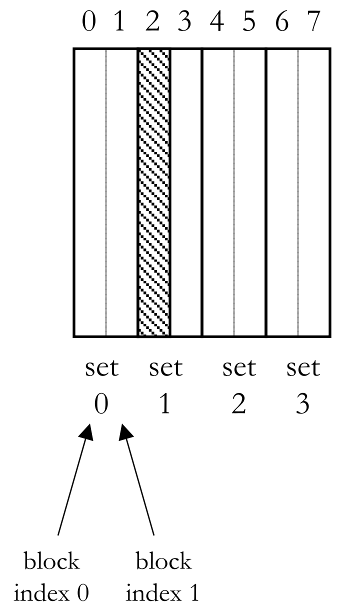
- it's a simple function: $f(A) = [2^n(A/2^n)] \bmod(\#sets)$
- $2^n(A/2^n)$ is the block number; this is integer division

The cache could be direct mapped (each set has a single block); or k-way set associative (each set has k blocks)

Summary So Far

In general, many memory blocks will map to each set in the cache

- in a direct-mapped cache, there's only room for one of these blocks at a time to be in a set
- in a set-associative cache, more than one of these blocks can stay in the cache at the same time, since each set has more than a single block in it



2-way set associative: each set can hold two memory blocks that map to that set

- if we want to read from a third memory block that maps to that set, then one of the blocks in that set is going to have to go
- the decision of which block to evict will depend on the replacement strategy

Terminology: the specific location of a block in a set is the *block index*

- block offset: where am I in a particular block?
- index: which set am I in?
- block index: which slot am I in in a particular set?

Question 3: Which Block Should be Replaced on a Cache Miss?

When a miss occurs (i.e., the data from the requested address is not present in the cache), then:

- the requested data has to be brought into the cache
- but in order to do this, a block that is currently in the cache must be evicted

And so, the question is: which block to evict?

- in a direct-mapped cache, there is only one possibility, since each set has a single block in it; so if a new block maps to set i, then whatever was in set i must be evicted
- in a fully associative or set-associative cache, there is more than one candidate: there is more than one block in the set, and the new block can go anywhere in the set; but ultimately, one of the existing blocks in the target set must be evicted

Associative Caches: Replacement Strategies

1. Random

- pick a random block in the target set and throw it out; the goal here is to spread out the allocation
- simple to build in hardware

2. Least recently used (LRU)

- use the idea of locality: if recently used blocks are likely to be used again soon, then a block that hasn't been used recently is a good candidate for eviction
- complex to implement in hardware, and time-consuming to check

3. First in, first out (FIFO)

- implementing LRU is complex; a simpler implementation is to evict the <u>oldest</u> block

Associative Caches: Replacement Strategies

Both LRU and FIFO will require some kind of data structure in hardware

- a queue of some sort

For FIFO, when a block is <u>brought in to the cache</u>

- its tag goes at the back of the queue

For LRU, when a block is <u>referenced</u>

- its tag is put to the back of the queue

This way, the tag at the front of the queue will be the oldest (for FIFO) or least recently referenced (for LRU) block

- and will correspond to the block that should be thrown out during replacement

Tag Queue: Examples

Suppose we have a four-way associative cache, and suppose that this is a sequence of memory references to blocks in a particular set:

{95, 12, 78, 12, 45, 12, 23, 95, 12, 78, 23, 12}

This shows the tag queue for a FIFO replacement cache after each of these references:

1	95	[95, x, x, x]	7	23	[23, 45, 78, 12]	red are cache misses
2	12	[12, 95, x, x]	8	95	[95, 23, 45, 78]	
3	78	[78, 12, 95, x]	9	12	[12, 95, 23, 45]	
4	12	[78, 12, 95, x]	10	78	[78, 12, 95, 23]	
5	45	[45, 78, 12, 95]	11	23	[78, 12, 95, 23]	
6	12	[45, 78, 12, 95]	12	12	[78, 12, 95, 23]	

this is just a priority queue, with the priority as "first time referenced"

Tag Queue: Examples

Suppose we have a four-way associative cache, and suppose that this is a sequence of memory references to blocks in a particular set:

{95, 12, 78, 12, 45, 12, 23, 95, 12, 78, 23, 12}

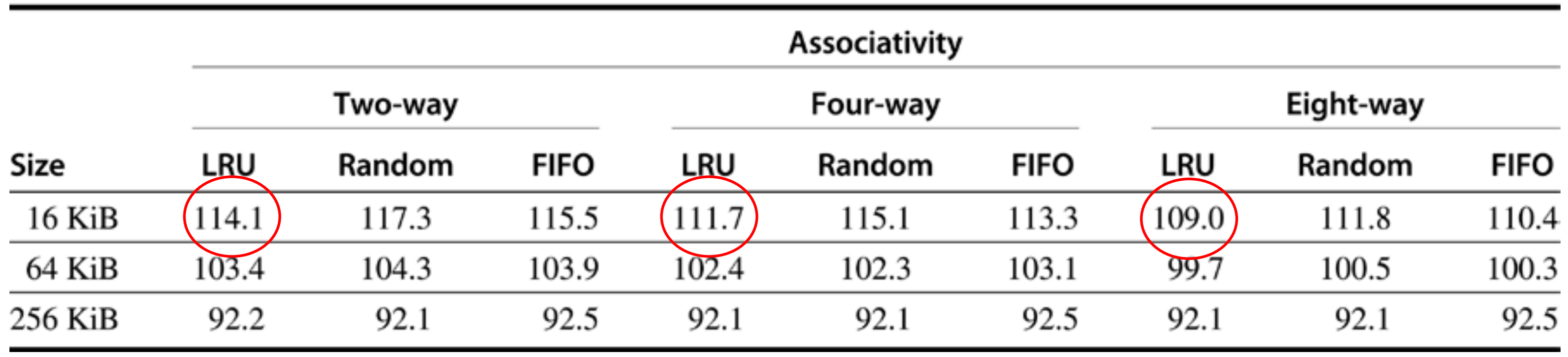
Same thing, with LRU (red are cache misses):

1	95	[95, x, x, x]	7	23	[23, 12, 45, 78]
2	12	[12, 95, x, x]	8	95	[95, 23, 12, 45]
3	78	[78, 12, 95, x]	9	12	[12, 95, 23, 45]
4	12	[12, 78, 95, x]	10	78	[78, 12, 95, 23]
5	45	[45, 12, 78, 95]	11	23	[23, 78, 12, 95]
6	12	[12, 45, 78, 95]	12	12	[12, 23, 78, 95]

In this example, LRU is better than FIFO (one fewer cache miss)

this is just a priority queue, with the priority as "most recent time referenced"

Example Results



	Associativity								
	Two-way			Four-way			Eight-way		
Size	LRU	Random	FIFO	LRU	Random	FIFO	LRU	Random	FIFO
16 KiB	114.1	117.3	115.5	111.7	115.1	113.3	109.0	111.8	110.4
64 KiB	103.4	104.3	103.9	102.4	102.3	103.1	99.7	100.5	100.3
256 KiB	92.2	92.1	92.5	92.1	92.1	92.5	92.1	92.1	92.5

Figure B.4: Data cache misses per 1000 instructions

For larger cache sizes, LRU and random perform about the same

For smaller caches, LRU outperforms the others

Q4: What Happens on a Write?

Most memory accesses are reads

- in particular, all instruction accesses are reads (think about this!)
- all instructions that do not involve a write are reads
- estimate from the book: 93% of all memory traffic consists of reads

Implication for design

- make the common case fast
- and in fact, the memory system can read a cache block at the same time that it's checking the tag
- if it's a hit, then the data is already available
- if it's a miss, then the data is discarded, and the only penalty was the extra read

Cache Writes

By contrast, the system can only write to a cache block if it is a hit

- and the size of the write (from one byte to n bytes) can vary as well

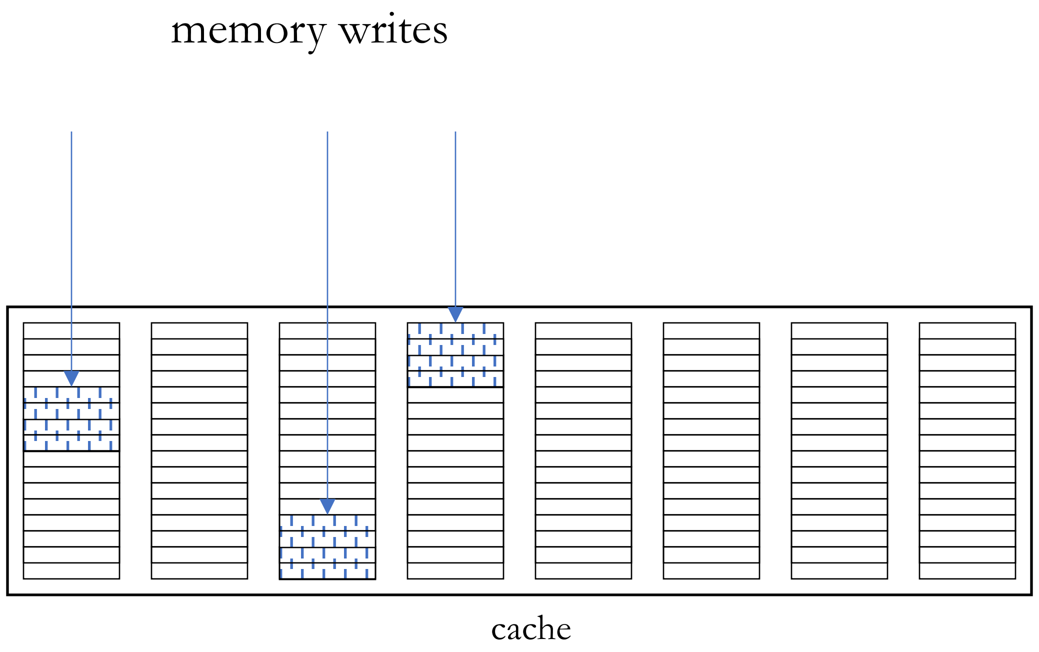
Two schemes for a write hit:

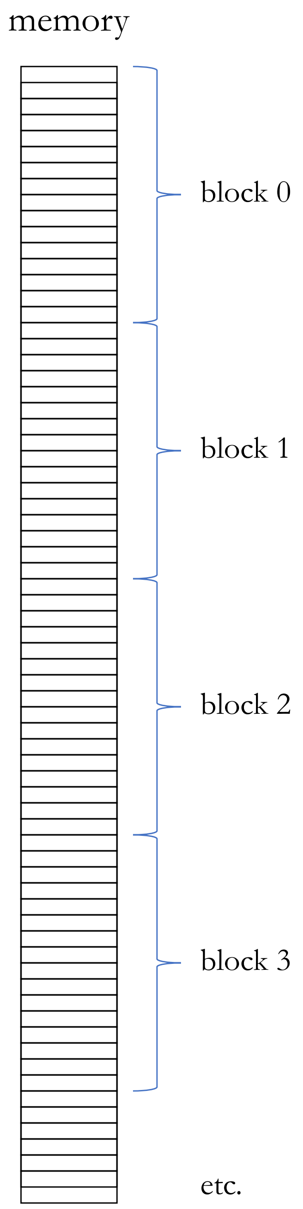
- **write through**: write to the cache and to the corresponding block in the main memory
- **write back**: write to the cache; write the contents of the cache block to main memory only when the cache block is replaced; use a *dirty bit* to tell the system whether the block has changed since it was read into the cache

Cache Writes on a Write Hit

Write-back cache

- the cache has the most up-to-date values
- cache blocks that have been written to are dirty

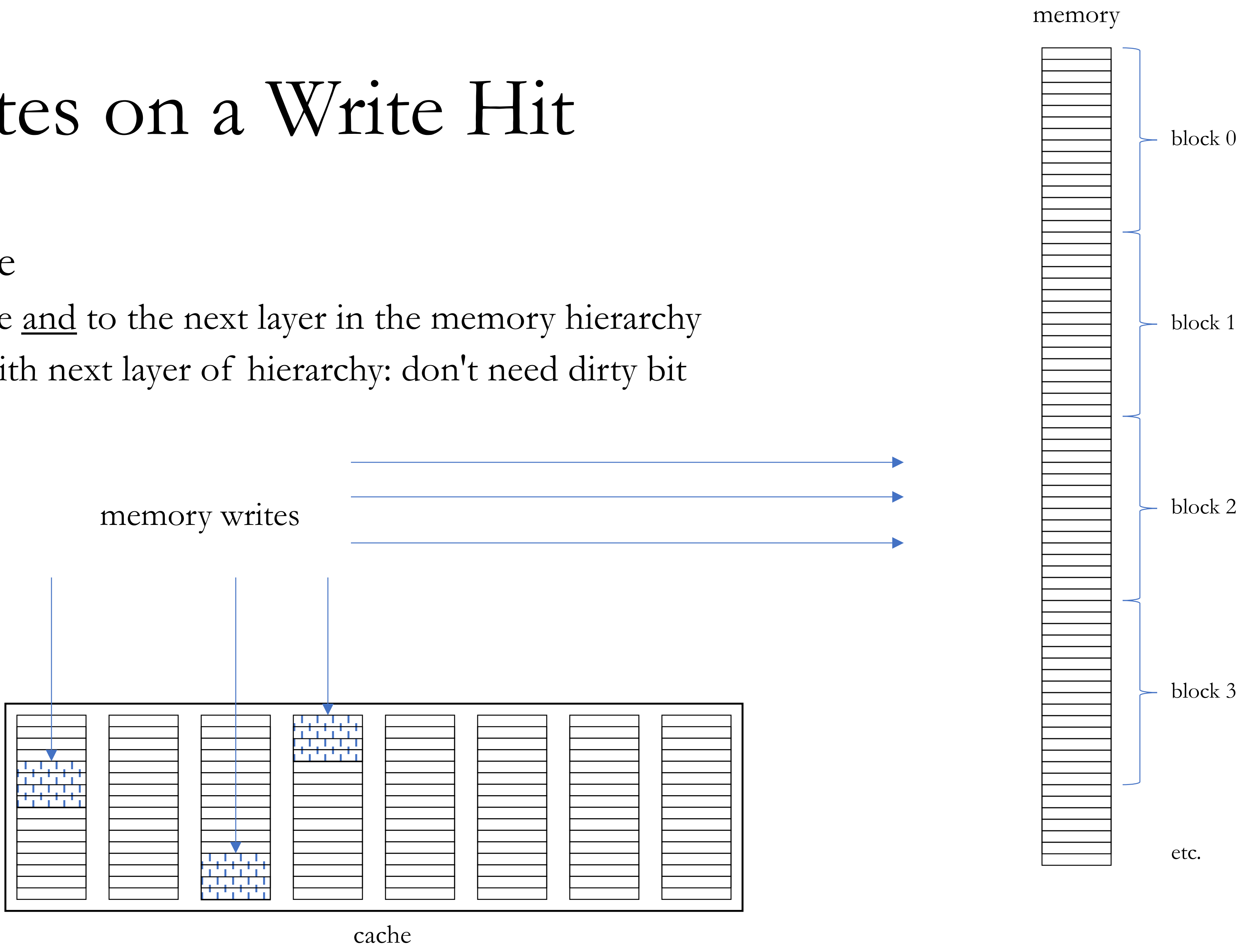




Cache Writes on a Write Hit

Write-through cache

- writes go to the cache <u>and</u> to the next layer in the memory hierarchy
- cache stays in sync with next layer of hierarchy: don't need dirty bit



Cache Writes

Write through

- keeps the data in the cache clean
- keeps the data at the next level of the hierarchy up to date—especially important for multicore processors and I/O
- simpler to implement
- slower: each write hit goes also to the next layer of hierarchy
- uses more memory bandwidth

Write back

- more complex
- faster: on a write hit, the write goes only to the top-level cache
- reduces memory traffic

Write Miss

What should happen during a write miss?

- different case from a read miss, since no data is actually returned

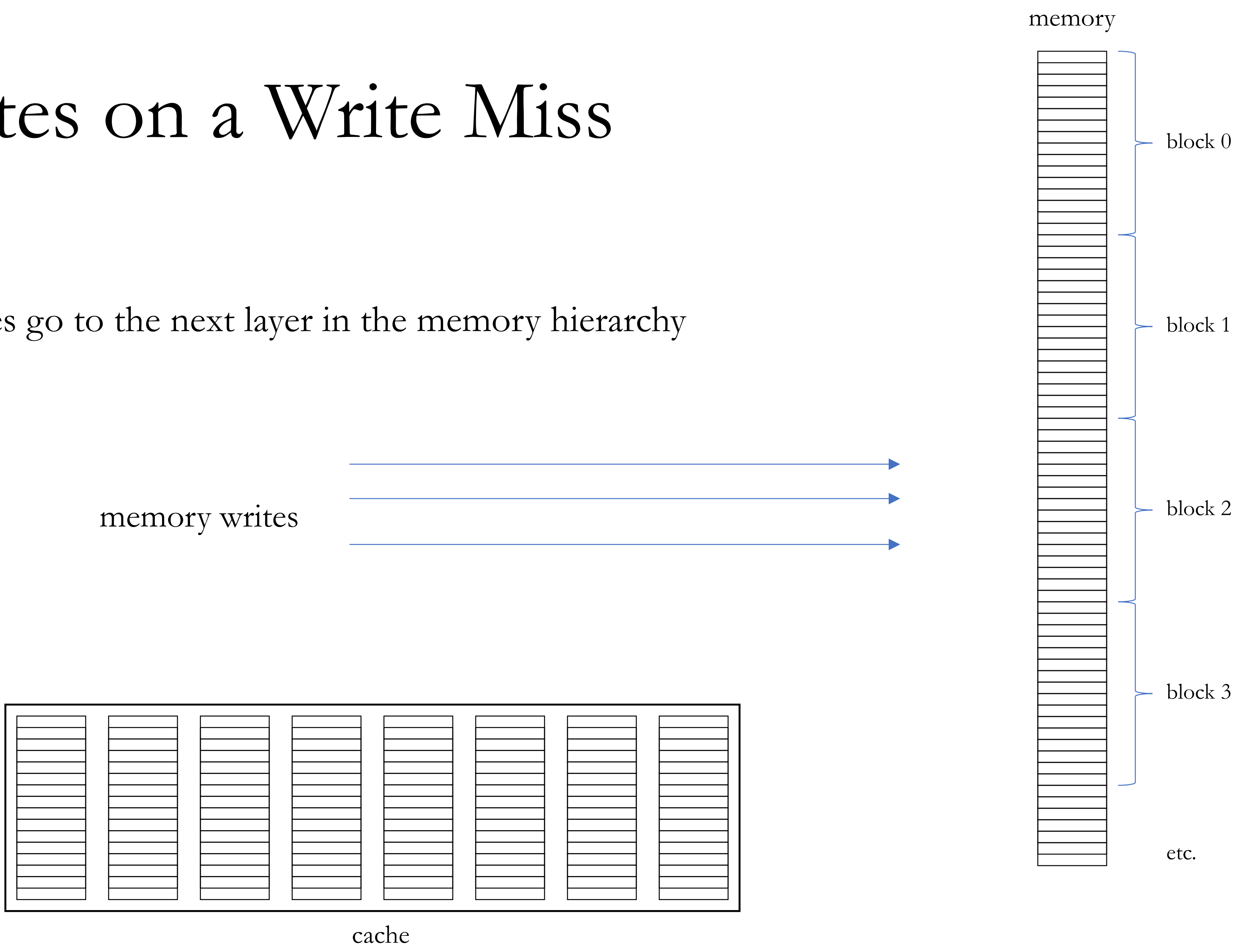
Two choices:

1. **write allocate**: the block is first allocated (read into the cache), and then the write occurs; so to the cache, this is like a read miss
2. **no-write allocate**: write misses do not affect the cache; instead, the block is modified only in the lower-level memory; these blocks stay out of the cache until requested during a subsequent read

Cache Writes on a Write Miss

No-write allocate

- on a write miss, writes go to the next layer in the memory hierarchy
- cache stays clean



Cache Writes on a Write Miss

Write allocate

- the target block is first brought into the cache on a write miss
- and then the actions for a write hit occur

Write Miss

Write-back caches typically use write allocate

- with the idea that there will be subsequent writes to the same location; and for those subsequent writes the block will already be in the cache

Write-through caches typically use no-write allocate

- since each cache write goes through to lower-level memory anyway, why cache the block?

Write Miss: Example

Consider this sequence of reads and writes to memory location X and memory location Y:

```
write to X
write to X
read from Y
write to Y
write to X
```

Compare and contrast the results using write allocate and no-write allocate schemes

Write Miss: Example

Consider this sequence of reads and writes to memory location X and memory location Y:

	write allocate	no-write allocate
`write to X`	miss	miss
`write to X`	hit	miss
`read from Y`	miss	miss
`write to Y`	hit	hit
`write to X`	hit	miss

- **write allocate**: the target memory block is first brought into the cache, and then the write occurs to the cache
- **no-write allocate**: the write occurs directly to memory and not to the cache

Cache Performance

To evaluate the performance of a cache, we could use miss rate

- but this really only provides incomplete information
- since it assumes that all cache misses have the same penalty
- it also assumes that all cache hits have the same latency

A better measure of performance is average memory access time

$$average\ memory\ access\ time = hit\ time + miss\ rate \times miss\ penalty$$

And we can specify hit time and miss penalty as absolute time values or as #cycles

Example

Design options: we have to decide which is better (in terms of smallest mean access time)

1) a single 32 KB unified cache (both instructions and data go into the same cache)
2) or separate 16 KB data cache and 16 KB instruction cache

Example

Statistics

- assume that 36% of instructions are data-transfer instructions
- a hit takes one clock cycle, and the miss penalty is 100 cycles
- a load or store hit takes one extra clock cycle on a unified cache (because of contention for a single read port)

Benchmark data

Size (KB)	**I-Cache**	**D-Cache**	**Unified Cache**
16	3.82	40.9	51.0
32	1.36	38.4	43.3

Figure B.4: Cache misses per 1000 instructions

A Step Back

To answer the question about the split caches vs. unified cache

- let’s take a step back
- and look at the role of memory references and instruction references during instruction processing

Instructions

Here is a sequence of ten RISC-V assembly-language instructions

```
lw    a0, -16(s0)   # load count
slli  a0, a0, 2     # count = count * 4
lw    a1, -24(s0)   # load i
slli  a1, a1, 2     # i = i * 4
bge   a1, a0, L1    # branch if a1 >= a0
lw    a2, -12(s0)   # load v
add   a2, a2, a1    # address of v[i]
lw    a0, 0(a2)     # contents of v[i]
lw    a1, -20(s0)   # load s
add   a1, a1, a0    # s = s + v[i]
```

The instructions in red involve data transfers

- data loads or stores (just loads, in this case)

Every instruction requires an access from instruction memory

- for the instruction itself

So for this sequence of 10 instructions

- there is a total of 10 + 5 = 15 memory references
- one third of the memory references are data references
- two thirds of the memory references are instruction references

Instruction Execution

Data for instruction execution comes from memory

- instructions come from one place in memory
- data comes from a different place in memory

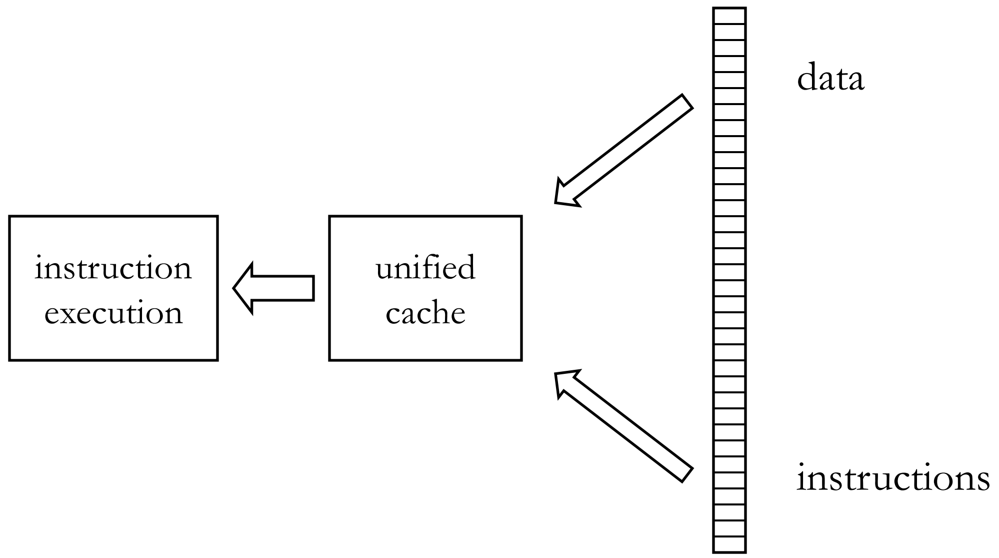
And so the question here is:

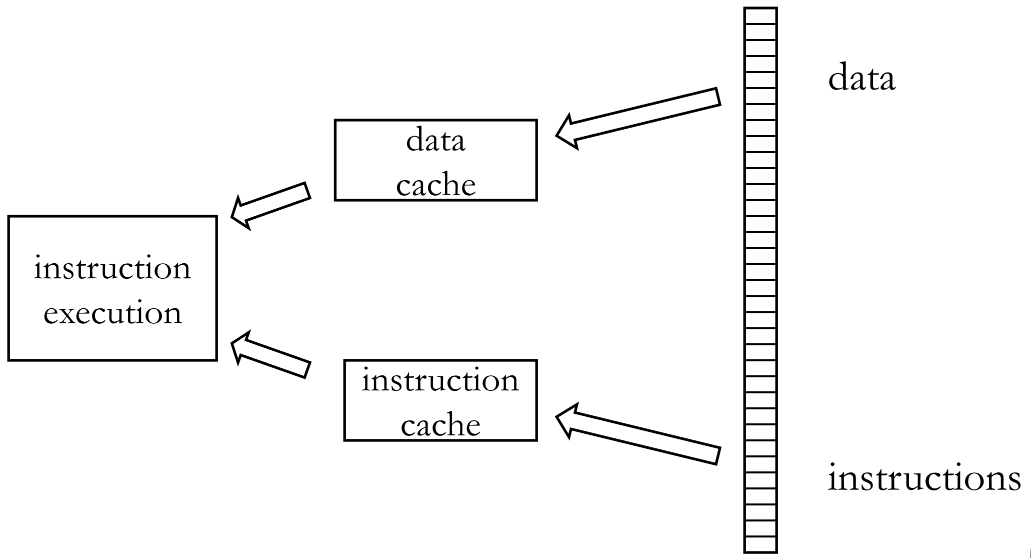
- should instructions and data both go into the same first-level cache?
- or should there be two different caches?

The Two Alternatives

Here are the two choices

- the actual amount of cache storage is the same
- it's just a question of whether there is a single larger cache or two smaller caches





Necessary Calculations

(1) convert misses per 1000 instructions to miss rate

miss rate: percentage of memory references that miss in the cache

For instructions:

- every instruction is a memory access (for the instruction itself)
- so 3.82 misses per 1000 instructions means 3.82 misses per 1000 instruction memory references
- which we can round up to a miss rate of 0.4%, or 0.004

$$\boxed{miss\ rate = \frac{\frac{misses}{1000\ instr} \div 1000}{\frac{mem\ accesses}{instr}}}$$

Necessary Calculations

(2) convert misses per 1000 instructions to miss rate

miss rate: percentage of memory references that miss in the cache

For data:

- 36% of the instructions are data transfers
- there are 40.9 data misses per 1000 instructions
- this means there are 40.9 data misses in the 360 instructions that are data transfers
- and so the miss rate is 40.9 / 360 ≈ 0.114

$$\boxed{miss\ rate = \frac{\frac{misses}{1000\ instr} \div 1000}{\frac{mem\ accesses}{instr}}}$$

Necessary Calculations

(3) calculate the total miss rate for the split cache

miss rate: percentage of memory references that miss in the cache

For data and instructions:

- every instruction requires an instruction fetch, and the miss rate for instructions is 0.004%
- 36% of the instructions require a data transfer, which means that in 1000 instructions, there are 1000 + 360 total memory references
- this means that in 1000 instructions, 1000 / (1000 + 360) ≈ 0.74 is the proportion of memory references that are <u>instruction</u> references
- so 74% of the total memory references are instruction references, and 26% are data references
- and so the total miss rate for the split cache is $0.74 * \underset{\text{miss rate for instructions}}{0.004} + 0.26 * \underset{\text{miss rate for data}}{0.114} \approx 0.0326$

Necessary Calculations

(4) calculate the total miss rate for the unified cache

miss rate: percentage of memory references that miss in the cache

For data and instructions:

- 36% of the instructions require a data transfer, which means that in 1000 instructions, there are 1000 + 360 total memory references
- for the unified cache, there are 43.3 cache misses per 1000 instructions
- this means there are 43.3 cache misses per 1360 combined instruction references + data references
- so the total miss rate for the unified cache is 43.3 / 1360 ≈ 0.0318

Bringing It All Together

Average memory access time is given by:

$$\begin{aligned}average\ memory\ access\ time &= \%instr \times (hit\ time + instr\ miss\ rate \times miss\ penalty) \\ &\quad + \%data \times (hit\ time + data\ miss\ rate \times miss\ penalty)\end{aligned}$$

So for the split cache:

$$\begin{aligned}avg\ mem\ access\ time &= 74\% \times (1 + \overset{\text{miss rate for instructions}}{0.004} \times 100) + 26\% \times (1 + \overset{\text{miss rate for data}}{0.114} \times 100) \\ &= (74\% \times 1.4) + (26\% \times 12.4) \\ &= 4.26\ cycles\end{aligned}$$

And for the unified cache:

$$\begin{aligned}avg\ mem\ access\ time &= 74\% \times (1 + \overset{\text{miss rate for unified cache}}{0.0318} \times 100) + 26\% \times (1 + \underset{\text{extra one-cycle penalty for a load/store hit}}{1} + \overset{\text{miss rate for unified cache}}{0.0318} \times 100) \\ &= (74\% \times 4.18) + (26\% \times 5.18) \\ &= 4.44\ cycles\end{aligned}$$

Conclusion

The split cache is better

- the unified cache has a bottleneck: only a single read or write per cycle

Effect of Cache Misses on CPU Performance

What we really care about is: how does the performance of the cache affect the performance of the CPU?

- specifically, how does a memory stall affect CPU time?

$$CPU\ time = IC \times \left(CPI_{execution} + \frac{memory\ stall\ clock\ cycles}{instruction} \right) \times cycle\ time$$

Then, for example, with cache miss penalty of 200 cycles, average miss rate of 2%, average of 1.5 memory references per instruction: there will be an average of 30 cache misses per 1000 instructions

$$\begin{aligned} CPU\ time &= IC \times \left(1.0 + \frac{30}{1000} \times 200 \right) \times cycle\ time \\ &= IC \times 7.0 \times cycle\ time \end{aligned}$$

In other words:

- there is a 7x penalty for the cache misses!

Miss Rate and CPU Performance

In terms of miss rate:

$$CPU\ time = IC \times \left(CPI_{execution} + miss\ rate \times \frac{memory\ accesses}{instruction} \times miss\ penalty\right) \times cycle\ time$$

$$= IC \times [1.0 + (1.5 \times 2\% \times 200)] \times\ cycle\ time$$

$$= IC \times 7.0 \times cycle\ time$$

And note: if there were no cache at all

- then <u>every</u> memory reference would incur the 200-cycle penalty
- and the CPU time would be $IC \times 301 \times cycle\ time$
- which is a 40x penalty compared to a system that has a cache!

Optimizing Cache Performance

We can increase the effectiveness of the cache in three ways:

1. by reducing the miss rate
2. by reducing the miss penalty
3. by reducing the time to hit in the cache

Because remember:

Average memory access time is given by:

$$\begin{aligned} \mathit{average\ memory\ access\ time} &= \%\mathit{instr} \times (\mathit{hit\ time} + \mathit{instr\ miss\ rate} \times \mathit{miss\ penalty}) \\ &+ \%\mathit{data} \times (\mathit{hit\ time} + \mathit{data\ miss\ rate} \times \mathit{miss\ penalty}) \end{aligned}$$

1. Increase the Block Size to Reduce Miss Rate

Advantage:

- larger block size means fewer misses
- larger block size reduces compulsory misses (a miss because the desired block is not in the cache)

Disadvantage

- increases this miss penalty (takes more time to read a larger block)
- increases conflict misses, since larger blocks mean that there are fewer blocks in the cache (the cache is finite in size!)

And at some point, we reach the point of diminishing returns

- when the average memory access time actually increases

2. Increase the Cache Size to Reduce Miss Rate

Advantage

- reduces the miss rate

Disadvantages

- higher cost and power and area (the area of a microprocessor is constrained)
- potentially longer hit time, since it takes longer to search through the cache for a desired block

3. Higher Associativity to Reduce Miss Rate

Advantage:

- reduces the miss rate

Disadvantage:

- increases hit time, since it takes longer to search through the tags in a target set

4. Multilevel Caches to Reduce Miss Penalty

Context:

- processor performance has scaled better than DRAM (main memory) performance
- meaning that the relative cost of a cache miss has become greater

Solution

- increase the effective cache size by using a multilevel cache (typically called L1, L2, L3, etc.)
- make L1 small to match the clock cycle time
- make L2 larger to capture many accesses that would otherwise go to main memory

Disadvantage

- complexity in design and analysis

5. Give Priority to Read Misses over Write Misses

As a way to reduce the miss penalty

The justification is somewhat complex

6. Avoid Address Translation during Indexing

Somewhat complex; has to do with virtual vs. physical addresses and page protection

- a key question: should we use virtual addresses or physical addresses for cache lookup?
- in fact, the question is: should we use a virtual or physical address to index the cache, and should we use a virtual or physical address in the tag comparison?

Virtually-addressed cache: faster

- no page translation required in order to do a cache lookup

However:

- memory protection is checked (and enforced) during page-table translation
- solution: include protection information in the cache
- another issue: the physical address that corresponds to a given virtual address can change during the lifetime of a process