

# Caching 3

# last time

tag / index / offset

lookup in associative caches

replacement policies

- least recently used — best miss rate assuming locality

- random — simplest to implement

write policies:

- write-through versus write-back

- write-allocate versus write-no-allocate

hit time, miss penalty, miss rate

average memory access time (AMAT)

cache design tradeoffs

## making any cache look bad

1. access enough blocks, to fill the cache
2. access an additional block, replacing something
3. access last block replaced
4. access last block replaced
5. access last block replaced
- ...

but — typical real programs have **locality**

# cache optimizations

	miss rate	hit time	miss penalty
increase cache size	better	worse	—
increase associativity	better	worse	worse?
increase block size	depends	worse	worse
add secondary cache	—	—	better
write-allocate	better	—	worse?
writeback	???	—	worse?
LRU replacement	better	?	worse?

average time = hit time + miss rate × miss penalty

# cache optimizations by miss type

	capacity	conflict	compulsory
increase cache size	fewer misses	fewer misses	—
increase associativity	—	fewer misses	—
increase block size	—	more misses	fewer misses

(assuming other listed parameters remain constant)

# prefetching

seems like we can't really improve cold misses...

have to have a miss to bring value into the cache?

# prefetching

seems like we can't really improve cold misses...

have to have a miss to bring value into the cache?

solution: don't require miss: 'prefetch' the value before it's accessed

remaining problem: how do we know what to fetch?

## common access patterns

suppose recently accessed 16B cache blocks are at:

0x48010, 0x48020, 0x48030, 0x48040

guess what's accessed next



## common access patterns

suppose recently accessed 16B cache blocks are at:

0x48010, 0x48020, 0x48030, 0x48040

guess what's accessed next

common pattern with **instruction fetches** and **array accesses**

## prefetching idea

look for sequential accesses

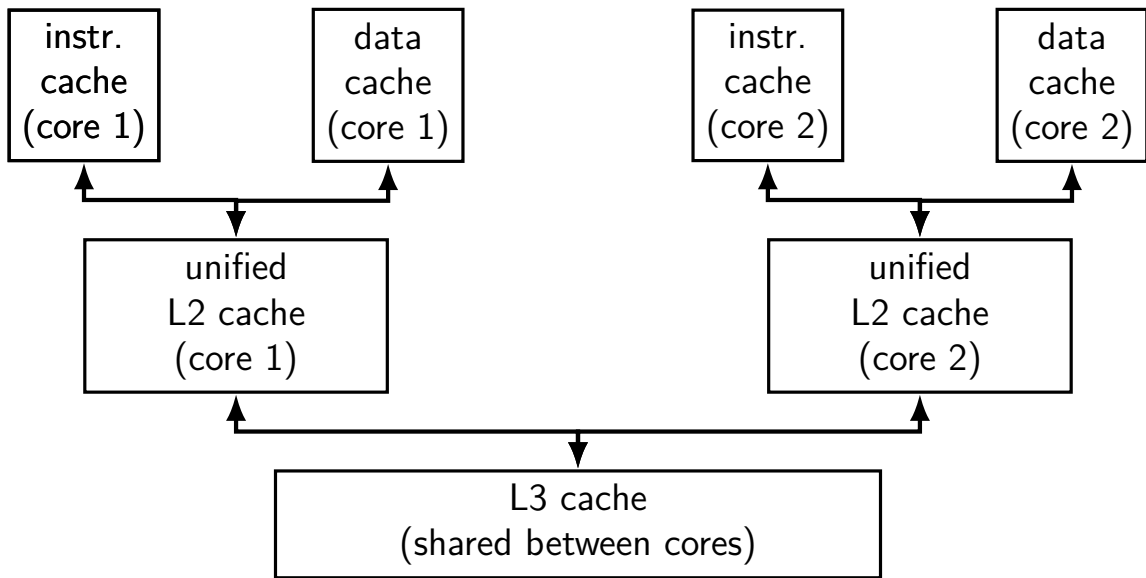
bring in guess at next-to-be-accessed value

if right: no cache miss (even if never accessed before)

if wrong: possibly evicted something else — could cause more misses

fortunately, sequential access guesses almost always right

## split caches; multiple cores



# hierarchy and instruction/data caches

typically separate data and instruction caches for L1

(almost) never going to read instructions as data or vice-versa

avoids instructions evicting data and vice-versa

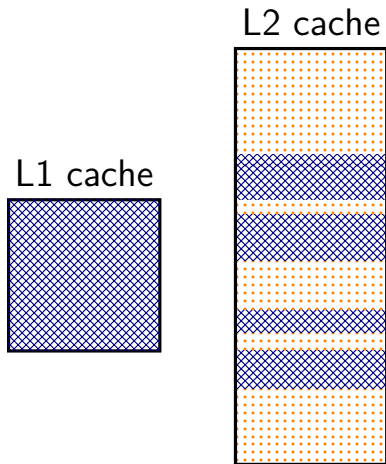
can optimize instruction cache for different access pattern

easier to build fast caches: that handles less accesses at a time

# inclusive versus exclusive

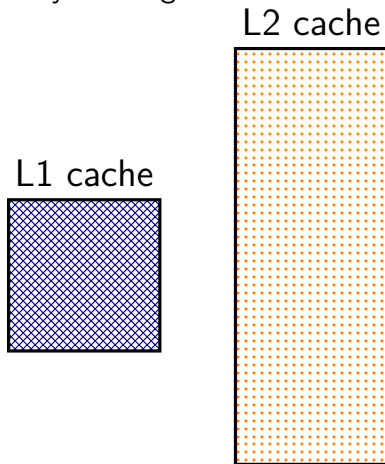
## L2 inclusive of L1

everything in L1 cache duplicated in L2  
adding to L1 also adds to L2



## L2 exclusive of L1

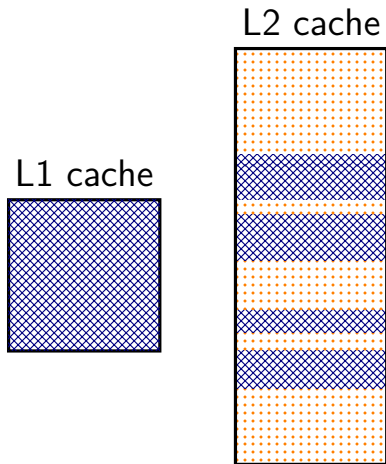
L2 contains different data than L1  
adding to L1 must remove from L2  
probably evicting from L1 adds to L2



# inclusive versus exclusive

## L2 inclusive of L1

everything in L1 cache duplicated in L2  
adding to L1 also adds to L2



## L2 exclusive of L1

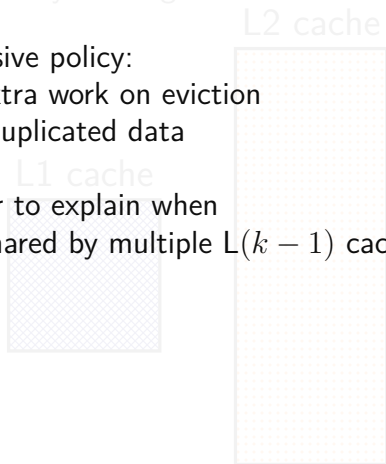
L2 contains different data than L1  
adding to L1 must remove from L2  
probably evicting from L1 adds to L2

inclusive policy:

no extra work on eviction  
but duplicated data

easier to explain when

$L_k$  shared by multiple  $L(k-1)$  caches?



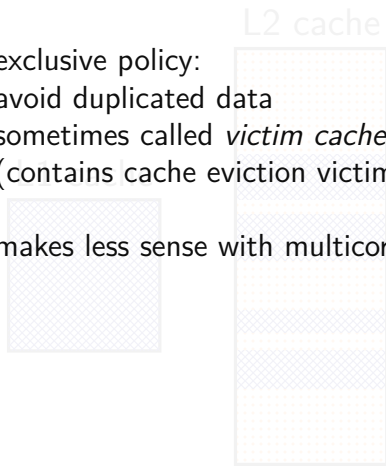
# inclusive versus exclusive

## L2 inclusive of L1

everything in L1 cache duplicated in L2  
adding to L1 also adds to L2

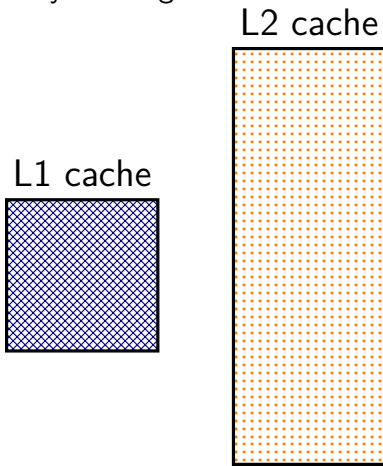
exclusive policy:  
avoid duplicated data  
sometimes called *victim cache*  
(contains cache eviction victims)

makes less sense with multicore



## L2 exclusive of L1

L2 contains different data than L1  
adding to L1 must remove from L2  
probably evicting from L1 adds to L2



## average memory access time

$AMAT = \text{hit time} + \text{miss penalty} \times \text{miss rate}$

effective speed of memory



# AMAT exercise (1)

90% cache hit rate

hit time is 2 cycles

30 cycle miss penalty

what is the average memory access time?

suppose we could increase hit rate by increasing its size, but it would increase the hit time to 3 cycles

how much do we have to increase the hit rate for this to be worthwhile?

# AMAT exercise (1)

90% cache hit rate

hit time is 2 cycles

30 cycle miss penalty

what is the average memory access time?

5 cycles

suppose we could increase hit rate by increasing its size, but it would increase the hit time to 3 cycles

how much do we have to increase the hit rate for this to be worthwhile?

# AMAT exercise (1)

90% cache hit rate

hit time is 2 cycles

30 cycle miss penalty

what is the average memory access time?

5 cycles

suppose we could increase hit rate by increasing its size, but it would increase the hit time to 3 cycles

how much do we have to increase the hit rate for this to be worthwhile?

to at least  $10\% - 1/30 \approx 94\%$

## exercise: AMAT and multi-level caches

suppose we have L1 cache with

- 3 cycle hit time

- 90% hit rate

and an L2 cache with

- 10 cycle hit time

- 80% hit rate (for accesses that make this far)

- (assume all accesses come via this L1)

and main memory has a 100 cycle access time

what is the average memory access time for the L1 cache?

## exercise: AMAT and multi-level caches

suppose we have L1 cache with

3 cycle hit time

90% hit rate

and an L2 cache with

10 cycle hit time

80% hit rate (for accesses that make this far)

(assume all accesses come via this L1)

and main memory has a 100 cycle access time

what is the average memory access time for the L1 cache?

$$3 + 0.1 \cdot (10 + 0.2 \cdot 100) = 6 \text{ cycles}$$

## exercise: AMAT and multi-level caches

suppose we have L1 cache with

3 cycle hit time

90% hit rate

and an L2 cache with

10 cycle hit time

80% hit rate (for accesses that make this far)

(assume all accesses come via this L1)

and main memory has a 100 cycle access time

what is the average memory access time for the L1 cache?

$$3 + 0.1 \cdot (10 + 0.2 \cdot 100) = 6 \text{ cycles}$$

$$\text{L1 miss penalty is } 10 + 0.2 \cdot 100 = 30 \text{ cycles}$$

## exercise (1)

initial cache: 64-byte blocks, 64 sets, 8 ways/set

If we leave the other parameters listed above unchanged, which will probably reduce the number of **capacity misses** in a typical program? (Multiple may be correct.)

- A. quadrupling the block size (256-byte blocks, 64 sets, 8 ways/set)
- B. quadrupling the number of sets
- C. quadrupling the number of ways/set

## exercise (2)

initial cache: 64-byte blocks, 8 ways/set, 64KB cache

If we leave the other parameters listed above unchanged, which will probably reduce the number of **capacity misses** in a typical program? (Multiple may be correct.)

- A. quadrupling the block size (256-byte block, 8 ways/set, 64KB cache)
- B. quadrupling the number of ways/set
- C. quadrupling the cache size



## exercise (3)

initial cache: 64-byte blocks, 8 ways/set, 64KB cache

If we leave the other parameters listed above unchanged, which will probably reduce the number of **conflict misses** in a typical program?  
(Multiple may be correct.)

- A. quadrupling the block size (256-byte block, 8 ways/set, 64KB cache)
- B. quadrupling the number of ways/set
- C. quadrupling the cache size

# cache accesses and C code (1)

```
int scaleFactor;
```

```
int scaleByFactor(int value) {  
    return value * scaleFactor;  
}
```

---

```
scaleByFactor:  
    movl scaleFactor, %eax  
    imull %edi, %eax  
    ret
```

---

exercice: what data cache accesses does this function do?

# cache accesses and C code (1)

```
int scaleFactor;
```

```
int scaleByFactor(int value) {  
    return value * scaleFactor;  
}
```

---

```
scaleByFactor:  
    movl scaleFactor, %eax  
    imull %edi, %eax  
    ret
```

---

exercice: what data cache accesses does this function do?

- 4-byte read of scaleFactor

- 8-byte read of return address

## possible scaleFactor use

```
for (int i = 0; i < size; ++i) {  
    array[i] = scaleByFactor(array[i]);  
}
```

## misses and code (2)

scaleByFactor:

```
movl scaleFactor, %eax
imull %edi, %eax
ret
```

suppose each time this is called in the loop:

return address located at address 0x7fffffe43b8

scaleFactor located at address 0x6bc3a0

with direct-mapped 32KB cache w/64 B blocks, what is their:

	return address	scaleFactor
tag		
index		
offset		

## misses and code (2)

scaleByFactor:

```
movl scaleFactor, %eax
imull %edi, %eax
ret
```

suppose each time this is called in the loop:

return address located at address 0x7fffffe43b8

scaleFactor located at address 0x6bc3a0

with direct-mapped 32KB cache w/64 B blocks, what is their:

	return address	scaleFactor
tag	0xffffffc	0xd7
index	0x10e	0x10e
offset	0x38	0x20

## misses and code (2)

scaleByFactor:

```
movl scaleFactor, %eax
imull %edi, %eax
ret
```

suppose each time this is called in the loop:

return address located at address 0x7fffffe43b8

scaleFactor located at address 0x6bc3a0

with direct-mapped 32KB cache w/64 B blocks, what is their:

	return address	scaleFactor
tag	0xffffffffc	0xd7
index	0x10e	0x10e
offset	0x38	0x20

## conflict miss coincidences?

obviously I set that up to have the same index

have to use exactly the right amount of stack space...

but gives a possible intuition for conflict misses:

bad luck giving the same index for unrelated values

matching experimental results:

most conflict misses involve a small portion of the sets



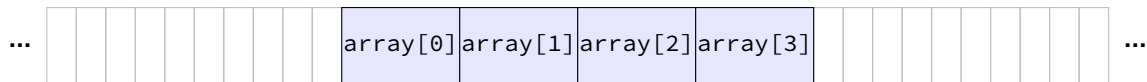
## C and cache misses (warmup 1)

```
int array[4];  
...  
int even_sum = 0, odd_sum = 0;  
even_sum += array[0];  
odd_sum += array[1];  
even_sum += array[2];  
odd_sum += array[3];
```

Assume everything but array is kept in registers (and the compiler does not do anything funny).

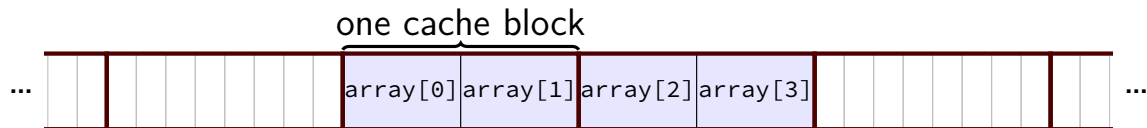
How many data cache misses on a 1-set direct-mapped cache with 8B blocks?

## some possibilities



Q1: how do cache blocks correspond to array elements?  
not enough information provided!

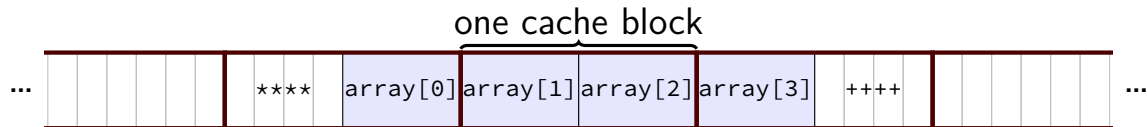
## some possibilities



if array[0] starts at beginning of a cache block...  
array split across two cache blocks

memory access	cache contents afterwards
—	(empty)
read array[0] (miss)	{array[0], array[1]}
read array[1] (hit)	{array[0], array[1]}
read array[2] (miss)	{array[2], array[3]}
read array[3] (hit)	{array[2], array[3]}

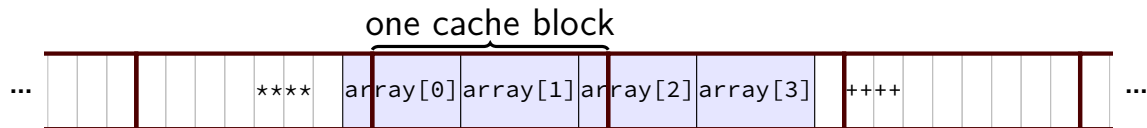
## some possibilities



if `array[0]` starts right in the middle of a cache block  
array split across three cache blocks

memory access	cache contents afterwards
—	(empty)
read <code>array[0]</code> (miss)	{****, <code>array[0]</code> }
read <code>array[1]</code> (miss)	{ <code>array[1]</code> , <code>array[2]</code> }
read <code>array[2]</code> (hit)	{ <code>array[1]</code> , <code>array[2]</code> }
read <code>array[3]</code> (miss)	{ <code>array[3]</code> , ++++}

# some possibilities



if array[0] starts at an odd place in a cache block,  
need to read two cache blocks to get most array elements

memory access	cache contents afterwards
—	(empty)
read array[0] byte 0 (miss)	{ ****, array[0] byte 0 }
read array[0] byte 1-3 (miss)	{ array[0] byte 1-3, array[2], array[3] byte 0 }
read array[1] (hit)	{ array[0] byte 1-3, array[2], array[3] byte 0 }
read array[2] byte 0 (hit)	{ array[0] byte 1-3, array[2], array[3] byte 0 }
read array[2] byte 1-3 (miss)	{ part of array[2], array[3], ++++ }
read array[3] (hit)	{ part of array[2], array[3], ++++ }

## aside: alignment

compilers and malloc/new implementations usually try **align** values

align = make address be multiple of something

most important reason: don't cross cache block boundaries

## C and cache misses (warmup 2)

```
int array[4];  
int even_sum = 0, odd_sum = 0;  
even_sum += array[0];  
even_sum += array[2];  
odd_sum += array[1];  
odd_sum += array[3];
```

Assume everything but array is kept in registers (and the compiler does not do anything funny).

Assume array[0] at beginning of cache block.

How many data cache misses on a 1-set direct-mapped cache with 8B blocks?

## C and cache misses (warmup 3)

```
int array[8];  
...  
int even_sum = 0, odd_sum = 0;  
even_sum += array[0];  
odd_sum += array[1];  
even_sum += array[2];  
odd_sum += array[3];  
even_sum += array[4];  
odd_sum += array[5];  
even_sum += array[6];  
odd_sum += array[7];
```

Assume everything but array is kept in registers (and the compiler does not do anything funny).

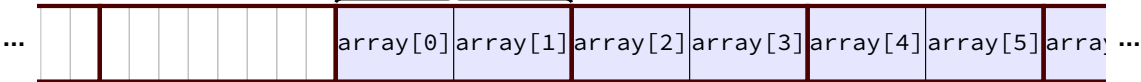
Assume array[0] at beginning of cache block.

How many data cache misses on a **2**-set direct-mapped cache with 8B blocks?

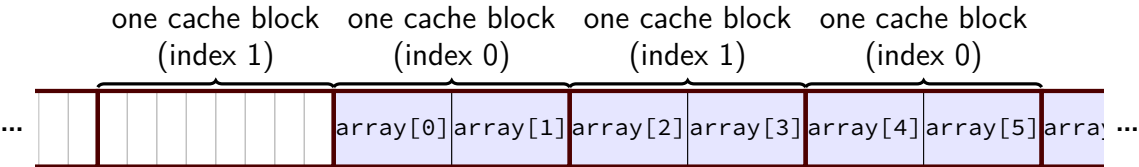


# exercise solution

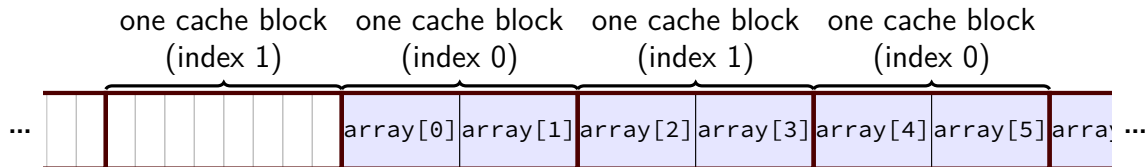
one cache block  
(index 0)



# exercise solution

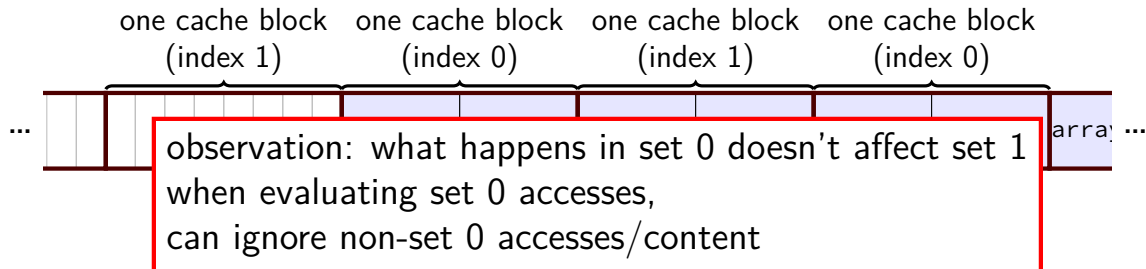


# exercise solution



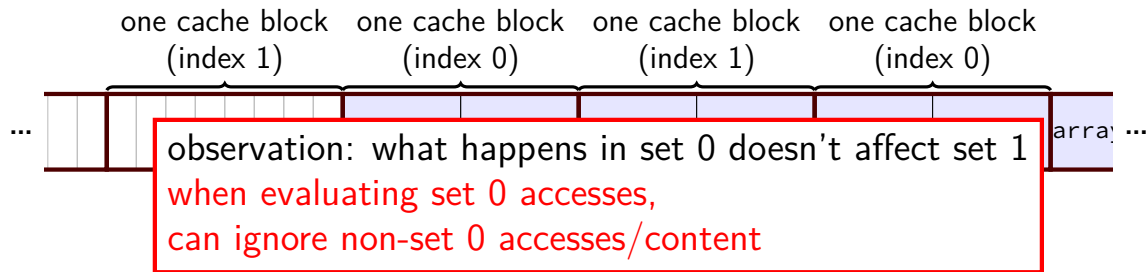
memory access	set 0 afterwards	set 1 afterwards
—	(empty)	(empty)
read array[0] (miss)	{array[0], array[1]}	(empty)
read array[1] (hit)	{array[0], array[1]}	(empty)
read array[2] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[3] (hit)	{array[0], array[1]}	{array[2], array[3]}
read array[4] (miss)	{array[4], array[5]}	{array[2], array[3]}
read array[5] (hit)	{array[4], array[5]}	{array[2], array[3]}
read array[6] (miss)	{array[4], array[5]}	{array[6], array[7]}
read array[7] (hit)	{array[4], array[5]}	{array[6], array[7]}

# exercise solution



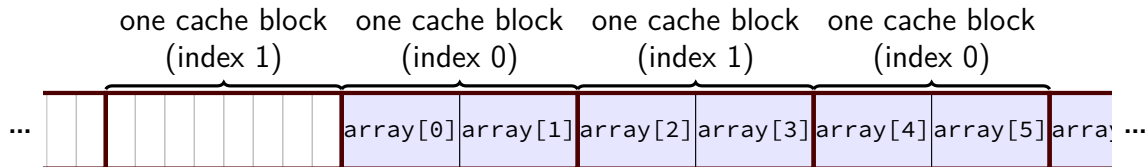
memory access	set 0 afterwards	set 1 afterwards
—	(empty)	(empty)
read array[0] (miss)	{array[0], array[1]}	(empty)
read array[1] (hit)	{array[0], array[1]}	(empty)
read array[2] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[3] (hit)	{array[0], array[1]}	{array[2], array[3]}
read array[4] (miss)	{array[4], array[5]}	{array[2], array[3]}
read array[5] (hit)	{array[4], array[5]}	{array[2], array[3]}
read array[6] (miss)	{array[4], array[5]}	{array[6], array[7]}
read array[7] (hit)	{array[4], array[5]}	{array[6], array[7]}

# exercise solution



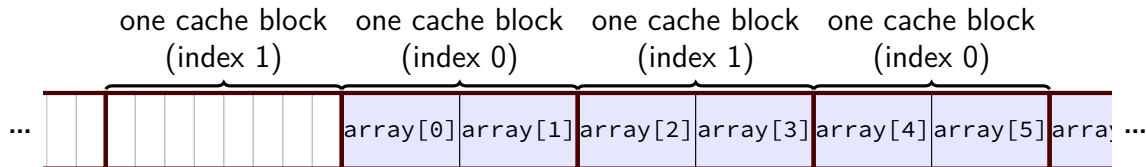
memory access	set 0 afterwards	set 1 afterwards
—	(empty)	(empty)
read array[0] (miss)	{array[0], array[1]}	(empty)
read array[1] (hit)	{array[0], array[1]}	(empty)
read array[2] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[3] (hit)	{array[0], array[1]}	{array[2], array[3]}
read array[4] (miss)	{array[4], array[5]}	{array[2], array[3]}
read array[5] (hit)	{array[4], array[5]}	{array[2], array[3]}
read array[6] (miss)	{array[4], array[5]}	{array[6], array[7]}
read array[7] (hit)	{array[4], array[5]}	{array[6], array[7]}

# exercise solution



memory access	set 0 afterwards	set 1 afterwards
—	(empty)	(empty)
read array[0] (miss)	{array[0], array[1]}	(empty)
read array[1] (hit)	{array[0], array[1]}	(empty)
read array[2] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[3] (hit)	{array[0], array[1]}	{array[2], array[3]}
read array[4] (miss)	{array[4], array[5]}	{array[2], array[3]}
read array[5] (hit)	{array[4], array[5]}	{array[2], array[3]}
read array[6] (miss)	{array[4], array[5]}	{array[6], array[7]}
read array[7] (hit)	{array[4], array[5]}	{array[6], array[7]}

# exercise solution



memory access	set 0 afterwards	set 1 afterwards
—	(empty)	(empty)
read array[0] (miss)	{array[0], array[1]}	(empty)
read array[1] (hit)	{array[0], array[1]}	(empty)
read array[2] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[3] (hit)	{array[0], array[1]}	{array[2], array[3]}
read array[4] (miss)	{array[4], array[5]}	{array[2], array[3]}
read array[5] (hit)	{array[4], array[5]}	{array[2], array[3]}
read array[6] (miss)	{array[4], array[5]}	{array[6], array[7]}
read array[7] (hit)	{array[4], array[5]}	{array[6], array[7]}

## C and cache misses (warmup 4)

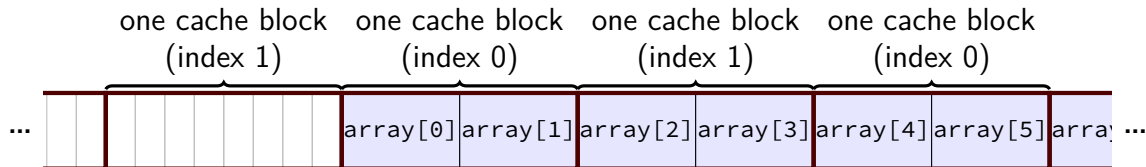
```
int array[8];  
...  
int even_sum = 0, odd_sum = 0;  
even_sum += array[0];  
even_sum += array[2];  
even_sum += array[4];  
even_sum += array[6];  
odd_sum += array[1];  
odd_sum += array[3];  
odd_sum += array[5];  
odd_sum += array[7];
```

Assume everything but array is kept in registers (and the compiler does not do anything funny).

How many data cache misses on a **2**-set direct-mapped cache with 8B blocks?

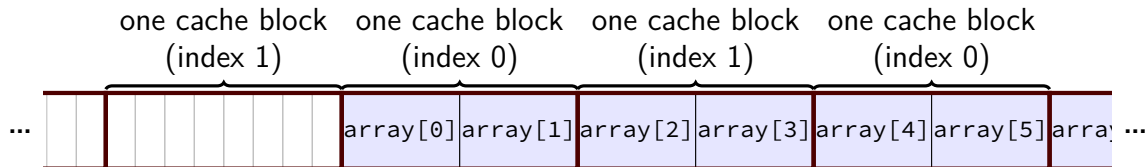


# exercise solution



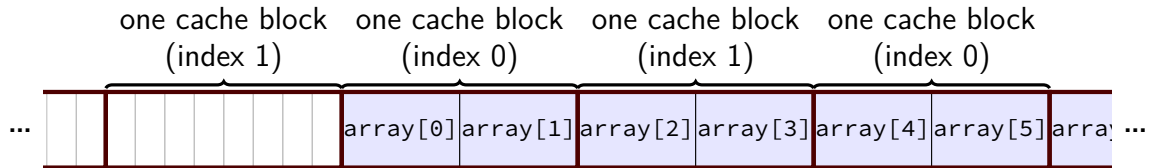
memory access	set 0 afterwards	set 1 afterwards
—	(empty)	(empty)
read array[0] (miss)	{array[0], array[1]}	(empty)
read array[2] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[4] (miss)	{array[4], array[5]}	{array[2], array[3]}
read array[6] (miss)	{array[4], array[5]}	{array[6], array[7]}
read array[1] (miss)	{array[0], array[1]}	{array[6], array[7]}
read array[3] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[5] (miss)	{array[4], array[5]}	{array[2], array[3]}
read array[7] (miss)	{array[4], array[5]}	{array[6], array[7]}

# exercise solution



memory access	set 0 afterwards	set 1 afterwards
—	(empty)	(empty)
read array[0] (miss)	{array[0], array[1]}	(empty)
read array[2] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[4] (miss)	{array[4], array[5]}	{array[2], array[3]}
read array[6] (miss)	{array[4], array[5]}	{array[6], array[7]}
read array[1] (miss)	{array[0], array[1]}	{array[6], array[7]}
read array[3] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[5] (miss)	{array[4], array[5]}	{array[2], array[3]}
read array[7] (miss)	{array[4], array[5]}	{array[6], array[7]}

# exercise solution



memory access	set 0 afterwards	set 1 afterwards
—	(empty)	(empty)
read array[0] (miss)	{array[0], array[1]}	(empty)
read array[2] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[4] (miss)	{array[4], array[5]}	{array[2], array[3]}
read array[6] (miss)	{array[4], array[5]}	{array[6], array[7]}
read array[1] (miss)	{array[0], array[1]}	{array[6], array[7]}
read array[3] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[5] (miss)	{array[4], array[5]}	{array[2], array[3]}
read array[7] (miss)	{array[4], array[5]}	{array[6], array[7]}

# arrays and cache misses (1)

```
int array[1024]; // 4KB array
int even_sum = 0, odd_sum = 0;
for (int i = 0; i < 1024; i += 2) {
    even_sum += array[i + 0];
    odd_sum += array[i + 1];
}
```

Assume everything but array is kept in registers (and the compiler does not do anything funny).

How many *data cache misses* on a 2KB direct-mapped cache with 16B cache blocks?

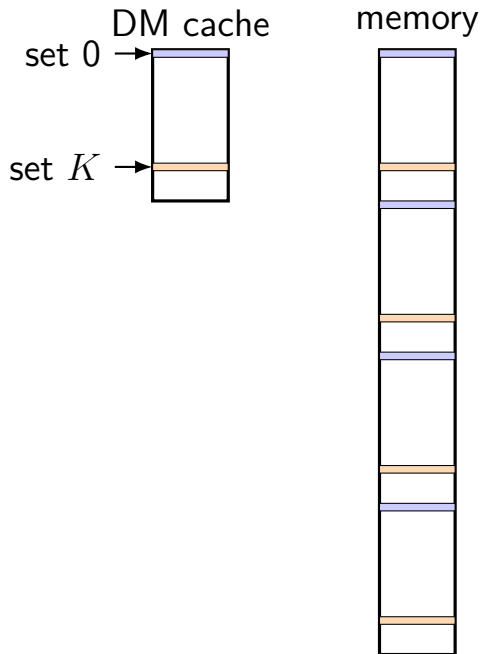
## arrays and cache misses (2)

```
int array[1024]; // 4KB array
int even_sum = 0, odd_sum = 0;
for (int i = 0; i < 1024; i += 2)
    even_sum += array[i + 0];
for (int i = 0; i < 1024; i += 2)
    odd_sum += array[i + 1];
```

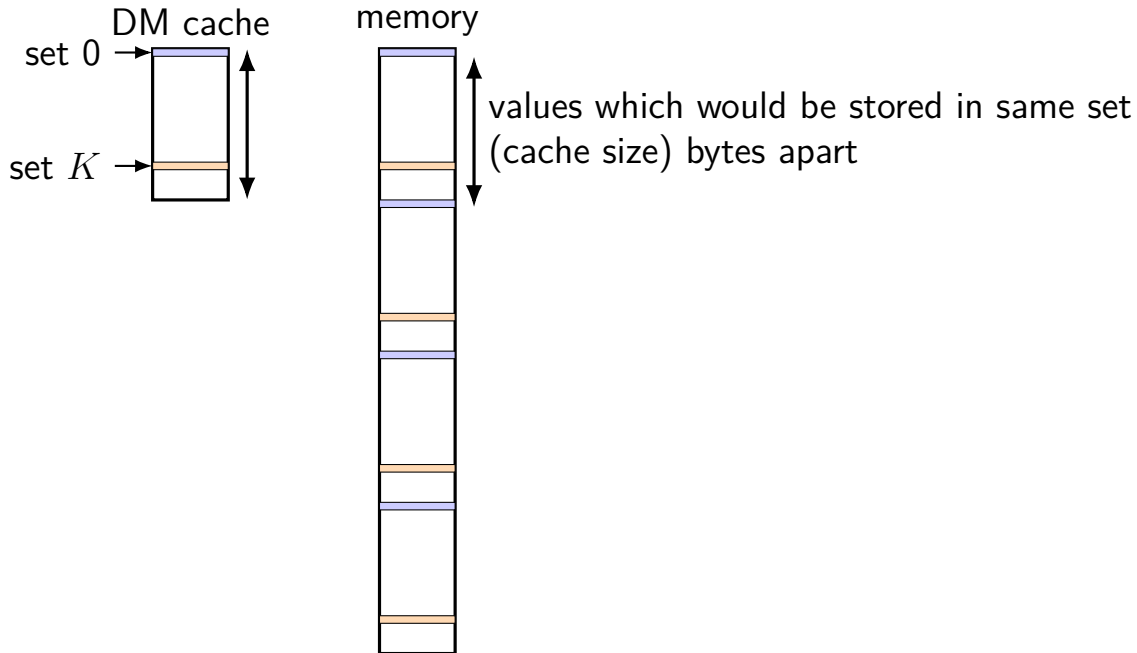
Assume everything but array is kept in registers (and the compiler does not do anything funny).

How many *data cache misses* on a 2KB direct-mapped cache with 16B cache blocks? Would a set-associative cache be better?

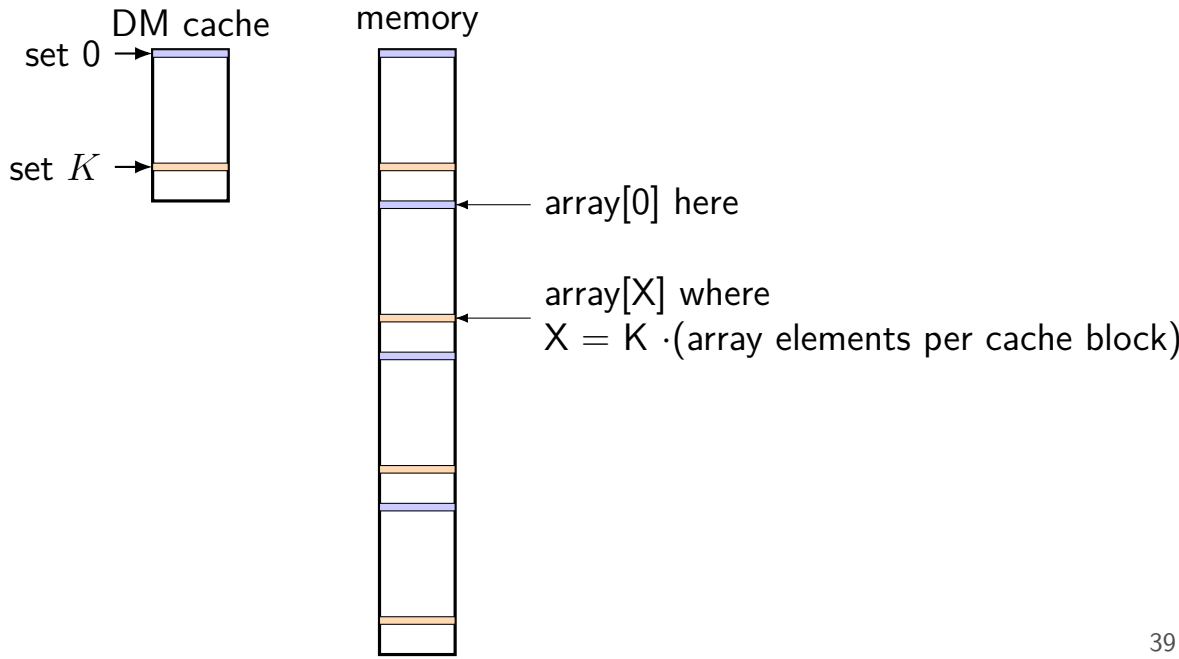
# mapping of sets to memory (direct-mapped)



# mapping of sets to memory (direct-mapped)

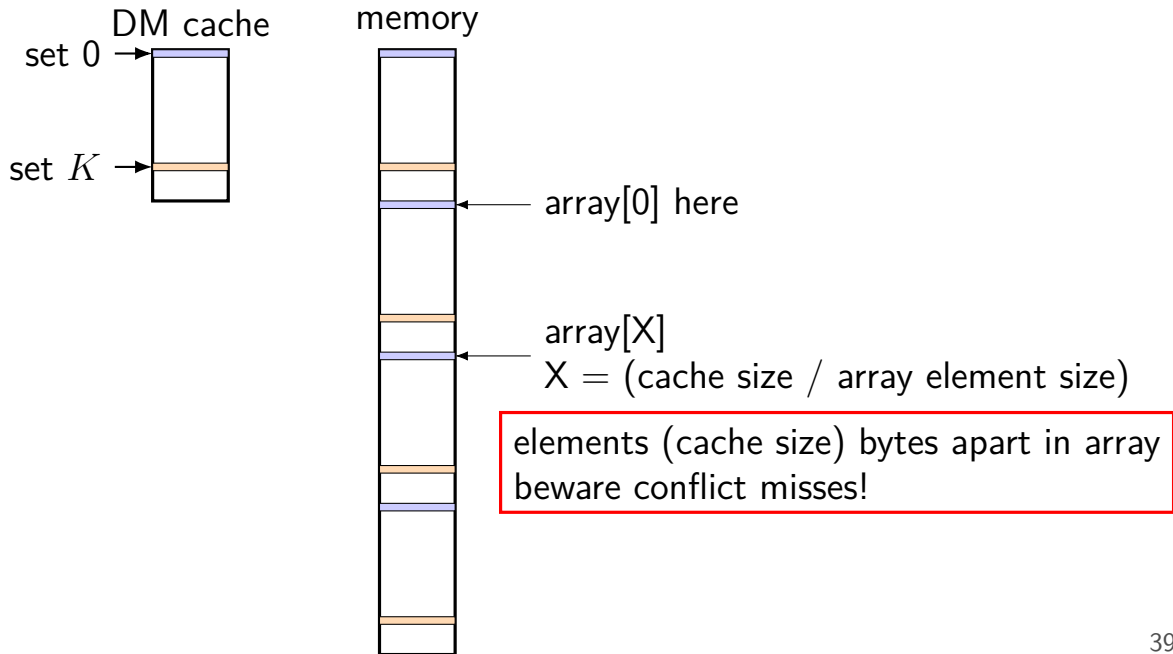


# mapping of sets to memory (direct-mapped)

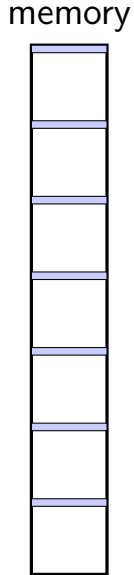
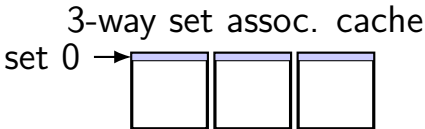




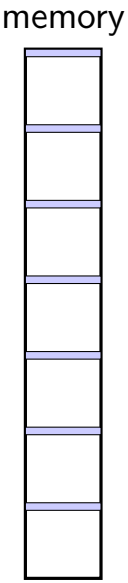
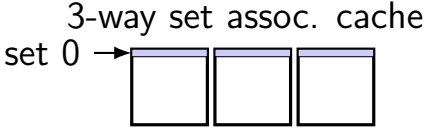
# mapping of sets to memory (direct-mapped)



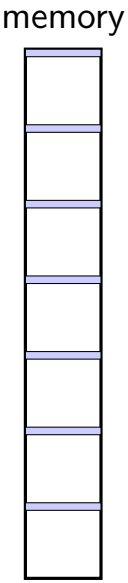
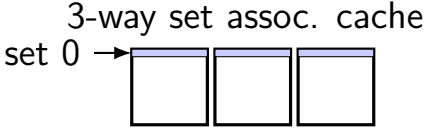
# mapping of sets to memory (3-way)



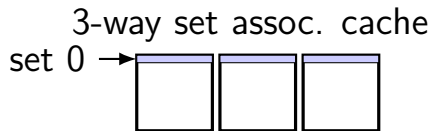
# mapping of sets to memory (3-way)



# mapping of sets to memory (3-way)



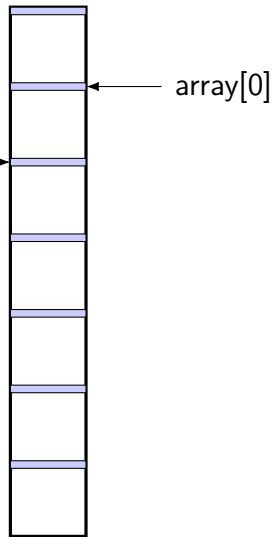
# mapping of sets to memory (3-way)



where  $X = \frac{\text{way size}}{\text{array element size}}$

accesses (way size) bytes apart in array?  
beware conflict misses!

memory



# thinking about cache storage (1)

2KB direct-mapped cache with 16B blocks —

set 0: address 0 to 15,  $(0 \text{ to } 15) + 2\text{KB}$ ,  $(0 \text{ to } 15) + 4\text{KB}$ , ...

set 1: address 16 to 31,  $(16 \text{ to } 31) + 2\text{KB}$ ,  $(16 \text{ to } 31) + 4\text{KB}$ , ...

...

set 127: address 2032 to 2047,  $(2032 \text{ to } 2047) + 2\text{KB}$ , ...

# thinking about cache storage (1)

2KB direct-mapped cache with 16B blocks —

set 0: address 0 to 15,  $(0 \text{ to } 15) + 2\text{KB}$ ,  $(0 \text{ to } 15) + 4\text{KB}$ , ...

set 1: address 16 to 31,  $(16 \text{ to } 31) + 2\text{KB}$ ,  $(16 \text{ to } 31) + 4\text{KB}$ , ...

...

set 127: address 2032 to 2047,  $(2032 \text{ to } 2047) + 2\text{KB}$ , ...

# thinking about cache storage (1)

2KB direct-mapped cache with 16B blocks —

set 0: address 0 to 15,  $(0 \text{ to } 15) + 2\text{KB}$ ,  $(0 \text{ to } 15) + 4\text{KB}$ , ...  
block at 0: array[0] through array[3]

set 1: address 16 to 31,  $(16 \text{ to } 31) + 2\text{KB}$ ,  $(16 \text{ to } 31) + 4\text{KB}$ , ...  
block at 16: array[4] through array[7]

...

set 127: address 2032 to 2047,  $(2032 \text{ to } 2047) + 2\text{KB}$ , ...  
block at 2032: array[508] through array[511]



# thinking about cache storage (1)

2KB direct-mapped cache with 16B blocks —

set 0: address 0 to 15,  $(0 \text{ to } 15) + 2\text{KB}$ ,  $(0 \text{ to } 15) + 4\text{KB}$ , ...

block at 0: `array[0]` through `array[3]`

block at  $0+2\text{KB}$ : `array[512]` through `array[515]`

set 1: address 16 to 31,  $(16 \text{ to } 31) + 2\text{KB}$ ,  $(16 \text{ to } 31) + 4\text{KB}$ , ...

block at 16: `array[4]` through `array[7]`

block at  $16+2\text{KB}$ : `array[516]` through `array[519]`

...

set 127: address 2032 to 2047,  $(2032 \text{ to } 2047) + 2\text{KB}$ , ...

block at 2032: `array[508]` through `array[511]`

block at  $2032+2\text{KB}$ : `array[1020]` through `array[1023]`

## thinking about cache storage (2)

2KB 2-way set associative cache with 16B blocks: block addresses

—

set 0: address 0,  $0 + 2\text{KB}$ ,  $0 + 4\text{KB}$ , ...

set 1: address 16,  $16 + 2\text{KB}$ ,  $16 + 4\text{KB}$ , ...

...

set 63: address 1008,  $2032 + 2\text{KB}$ ,  $2032 + 4\text{KB}$  ...

## thinking about cache storage (2)

2KB 2-way set associative cache with 16B blocks: block addresses

---

set 0: address 0,  $0 + 2\text{KB}$ ,  $0 + 4\text{KB}$ , ...  
block at 0: array[0] through array[3]

set 1: address 16,  $16 + 2\text{KB}$ ,  $16 + 4\text{KB}$ , ...  
address 16: array[4] through array[7]

...

set 63: address 1008,  $2032 + 2\text{KB}$ ,  $2032 + 4\text{KB}$  ...  
address 1008: array[252] through array[255]

## thinking about cache storage (2)

2KB 2-way set associative cache with 16B blocks: block addresses

---

set 0: address 0, 0 + 2KB, 0 + 4KB, ...

block at 0: array[0] through array[3]

block at 0+1KB: array[256] through array[259]

block at 0+2KB: array[512] through array[515]

...

set 1: address 16, 16 + 2KB, 16 + 4KB, ...

address 16: array[4] through array[7]

...

set 63: address 1008, 2032 + 2KB, 2032 + 4KB ...

address 1008: array[252] through array[255]

## thinking about cache storage (2)

2KB 2-way set associative cache with 16B blocks: block addresses

---

set 0: address 0,  $0 + 2\text{KB}$ ,  $0 + 4\text{KB}$ , ...

block at 0: `array[0]` through `array[3]`

block at  $0+1\text{KB}$ : `array[256]` through `array[259]`

block at  $0+2\text{KB}$ : `array[512]` through `array[515]`

...

set 1: address 16,  $16 + 2\text{KB}$ ,  $16 + 4\text{KB}$ , ...

address 16: `array[4]` through `array[7]`

...

set 63: address 1008,  $2032 + 2\text{KB}$ ,  $2032 + 4\text{KB}$  ...

address 1008: `array[252]` through `array[255]`

## C and cache misses (3)

```
typedef struct {
    int a_value, b_value;
    int boring_values[126];
} item;
item items[8]; // 4 KB array
int a_sum = 0, b_sum = 0;
for (int i = 0; i < 8; ++i)
    a_sum += items[i].a_value;
for (int i = 0; i < 8; ++i)
    b_sum += items[i].b_value;
```

Assume everything but `items` is kept in registers (and the compiler does not do anything funny).

How many *data cache misses* on a 2KB direct-mapped cache with 16B cache blocks?

## C and cache misses (3, rewritten?)

```
item array[1024]; // 4 KB array
int a_sum = 0, b_sum = 0;
for (int i = 0; i < 1024; i += 128)
    a_sum += array[i];
for (int i = 1; i < 1024; i += 128)
    b_sum += array[i];
```

## C and cache misses (4)

```
typedef struct {
    int a_value, b_value;
    int boring_values[126];
} item;
item items[8]; // 4 KB array
int a_sum = 0, b_sum = 0;
for (int i = 0; i < 8; ++i)
    a_sum += items[i].a_value;
for (int i = 0; i < 8; ++i)
    b_sum += items[i].b_value;
```

Assume everything but `items` is kept in registers (and the compiler does not do anything funny).

How many *data cache misses* on a 4-way set associative 2KB direct-mapped cache with 16B cache blocks?