last time (1)

replacement policies

- least recently used (LRU) — best for locality assumption
- approximations of LRU — more practical with > 2 ways
- first-in, first-out; random — much easier to implement, not that much worse

miss types:

- compulsory/cold: first access / conflict: fixed with more associativity / capacity: just not big enough

last time (2)

write-allocate versus write-no-allocate

- behavior if writing to not-yet-cached value
- write-allocate: add to cache
- write-no-allocate: don't add to cache

write-through versus write-back

- behavior if writing to value to be stored in cache
- if write-through, send writes immediately to next level
- if write-back, mark as dirty in cache, send to next level when evicted from cache

homework note

lab this week: cachelab (exercies)

homework: pipeline sim

- instruction trace: list of instructions run + src, dest, etc. as CSV file
- python-based "simulator" that counts cycles based on our processor design
- assignment: make some modifications

exercise (1)

2-way set associative, LRU, write-allocate, writeback

index	valid	tag	value	dirty	valid	tag	value	dirty	LRU
0	1	001100	mem[0x30] mem[0x31]	0	1	010000	mem[0x40]* mem[0x41]*	1	0
1	1	011000	mem[0x62] mem[0x63]	0	1	001100	mem[0x32]* mem[0x33]*	1	1

for each of the following accesses, performed alone, would it require (a) reading a value from memory (or next level of cache) and (b) writing a value to the memory (or next level of cache)?

writing 1 byte to 0x33
reading 1 byte from 0x52
reading 1 byte from 0x50

exercise (1, solution)

2-way set associative, LRU, write-allocate, writeback

index	valid	tag	value	dirty	valid	tag	value	dirty	LRU
0	1	001100	mem[0x30] mem[0x31]	0	1	010000	mem[0x40]* mem[0x41]*	1	0
1	1	011000	mem[0x62] mem[0x63]	0	1	001100	mem[0x32]* mem[0x33]*	~~1~~0	1

writing 1 byte to 0x33: (set 1, offset 1) no read or write

reading 1 byte from 0x52:

reading 1 byte from 0x50:

exercise (1, solution)

2-way set associative, LRU, write-allocate, writeback

index	valid	tag	value	dirty	valid	tag	value	dirty	LRU
0	1	001100	mem[0x30] mem[0x31]	0	1	010000	mem[0x40]* mem[0x41]*	1	0
1	1	011000	mem[0x62] mem[0x63]	0	1	001100	mem[0x50] mem[0x51]	~~0~~1	1

writing 1 byte to 0x33: (set 1, offset 1) no read or write

reading 1 byte from 0x52: (set 1, offset 0) **write** back 0x32-0x33; **read** 0x52-0x53

reading 1 byte from 0x50:

exercise (1, solution)

2-way set associative, LRU, write-allocate, writeback

index	valid	tag	value	dirty	valid	tag	value	dirty	LRU
0	1	001100	mem[0x30] mem[0x31]	0	1	010000	mem[0x40]* mem[0x41]*	1	0
1	1	011000	mem[0x62] mem[0x63]	0	1	001100	mem[0x32]* mem[0x33]*	1	1

writing 1 byte to 0x33: (set 1, offset 1) no read or write

reading 1 byte from 0x52: (set 1, offset 0) **write** back 0x32-0x33; **read** 0x52-0x53

reading 1 byte from 0x50: (set 0, offset 0) replace 0x30-0x31 (no write back); **read** 0x50-0x51

exercise (2)

2-way set associative, LRU, write-no-allocate, write-through

index	valid	tag	value	valid	tag	value	LRU
0	1	001100	mem[0x30] mem[0x31]	1	010000	mem[0x40] mem[0x41]	0
1	1	011000	mem[0x62] mem[0x63]	1	001100	mem[0x32] mem[0x33]	1

for each of the following accesses, performed alone, would it require (a) reading a value from memory and (b) writing a value to the memory?

writing 1 byte to 0x33
reading 1 byte from 0x52
reading 1 byte from 0x50

exercise (2, solution)

2-way set associative, LRU, write-no-allocate, write-through

index	valid	tag	value	valid	tag	value	LRU
0	1	001100	mem[0x30] mem[0x31]	1	010000	mem[0x40] mem[0x41]	0
1	1	011000	mem[0x62] mem[0x63]	1	001100	mem[0x32] mem[0x33]	~~1~~0

writing 1 byte to 0x33: (set 1, offset 1) write-through 0x33 modification

reading 1 byte from 0x52:

reading 1 byte from 0x50:

exercise (2, solution)

2-way set associative, LRU, write-no-allocate, write-through

index	valid	tag	value	valid	tag	value	LRU
0	1	001100	mem[0x50] mem[0x51]	1	010000	mem[0x40] mem[0x41]	~~01~~
1	1	011000	mem[0x62] mem[0x63]	1	001100	mem[0x52] mem[0x53]	~~10~~

writing 1 byte to 0x33: (set 1, offset 1) write-through 0x33 modification

reading 1 byte from 0x52: (set 1, offset 0) replace 0x32-0x33; **read** 0x52-0x53

reading 1 byte from 0x50:

exercise (2, solution)

2-way set associative, LRU, write-no-allocate, write-through

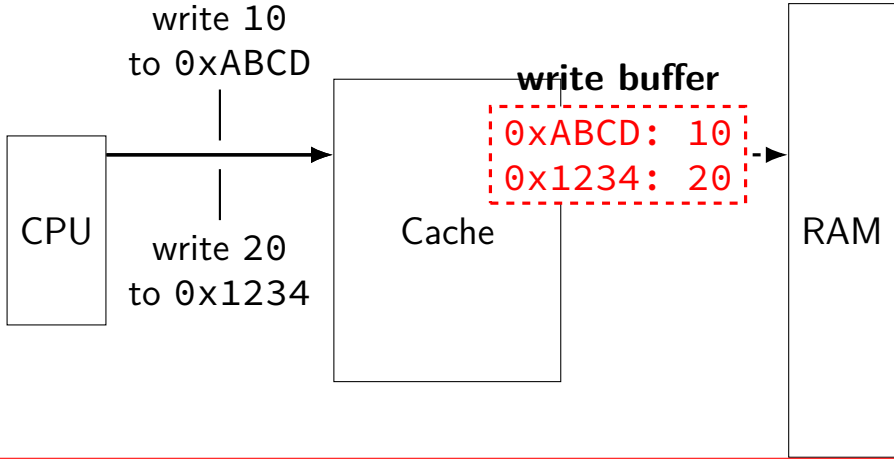
index	valid	tag	value	valid	tag	value	LRU
0	1	001100	mem[0x30] mem[0x31]	1	010000	mem[0x40] mem[0x41]	0
1	1	011000	mem[0x62] mem[0x63]	1	001100	mem[0x32] mem[0x33]	1

writing 1 byte to 0x33: (set 1, offset 1) write-through 0x33 modification

reading 1 byte from 0x52: (set 1, offset 0) replace 0x32-0x33; **read** 0x52-0x53

reading 1 byte from 0x50: (set 0, offset 0) replace 0x30-0x31; **read** 0x50-0x51

fast writes



write appears to complete immediately when placed in buffer
memory can be much slower

cache hierarchies

my desktop

4x L1 data cache per core
- 32KB, 8-way; write-back; 64B block

4x L1 instruction cache per core
- 32KB, 8-way; write-back; 64B block

4x L2 unified cache per core
- 256KB; 4-way; write-back; 64B block

1x L3 cache shared between all cores
- 8MB; 16-way; write-back; 64B block

13 caches total!

cache hierarchies

my desktop

4x L1 data cache per core
- 32KB, 8-way; write-back; 64B block

4x L1 instruction cache per core
- 32KB, 8-way; write-back; 64B block

4x L2 unified cache per core
- 256KB; 4-way; write-back; 64B block

1x L3 cache shared between all cores
- 8MB; 16-way; write-back; 64B block

13 caches total!

if something modified in one cache, how do others know?
- problem called cache coherency

average memory access time

AMAT = hit time + miss penalty × miss rate

or AMAT = hit time × hit rate + miss time × miss rate

effective speed of memory

AMAT example

suppose cache has 10 cycle hit time

80% hit rate

100 cycle miss penalty

AMAT example

suppose cache has 10 cycle hit time

80% hit rate

100 cycle miss penalty

AMAT = hit time + miss rate $\times$ miss penalty

AMAT = 10 cycles + (100% - 80%) $\times$ 100 cycles = 30 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

assume when there's an cache miss, the next level access starts after the hit time

- e.g. an access that misses in L1 and hits in L2 will take 10+3 cycles

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

assume when there's an cache miss, the next level access starts after the hit time

- e.g. an access that misses in L1 and hits in L2 will take 10+3 cycles

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

- $3 + 0.1 \cdot (10 + 0.2 \cdot 100) = 6$ 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

assume when there's an cache miss, the next level access starts after the hit time

- e.g. an access that misses in L1 and hits in L2 will take 10+3 cycles

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

- $3 + 0.1 \cdot (10 + 0.2 \cdot 100) = 6$ cycles
- L1 miss penalty is $10 + 0.2 \cdot 100 = 30$ cycles

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

(assuming typical locality...)

	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	—	?
writeback	—	—	?
LRU replacement	better	?	worse?
prefetching	better	—	—

prefetching = guess what program will use, access in advance

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

cache optimizations by miss type

(assuming other listed parameters remain constant)

	capacity	conflict	compulsory
increase cache size	fewer misses	fewer misses	—
increase associativity	—	fewer misses	—
increase block size	more misses?	more misses?	fewer misses
LRU replacement	—	fewer misses	—
prefetching	—	—	fewer misses

cache accesses and C code (1)

```
int scaleFactor;

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

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

---

exericse: 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
```

---

exericse: 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 `0x7ffffffe43b8`

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 `0x7ffffffe43b8`
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 `0x7ffffffe43b8`
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

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 one possible reason for conflict misses:

bad luck giving the same index for unrelated values

more direct reason: values related by power of two

- some examples later, probably

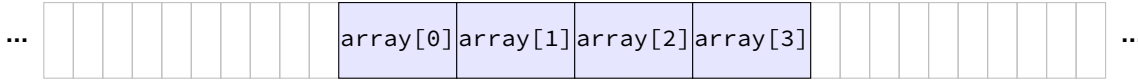
C and cache misses (warmup 1)

```
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).

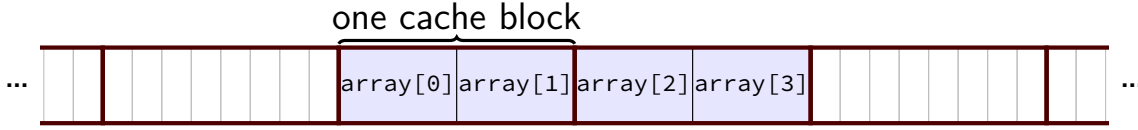
How many data cache misses on a 1-set direct-mapped cache with 8B blocks?

some possiblities



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

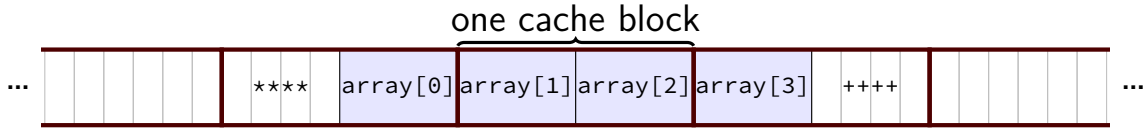
some possiblities



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]`}

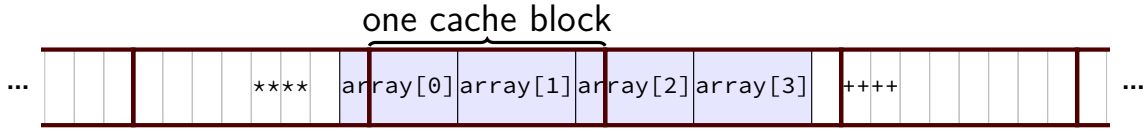
some possiblities



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 `array[0]` (miss)	{****, `array[0]`}
read `array[1]` (miss)	{`array[1]`, `array[2]`}
read `array[2]` (hit)	{`array[1]`, `array[2]`}
read `array[3]` (miss)	{`array[3]`, ++++}

some possiblities



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)

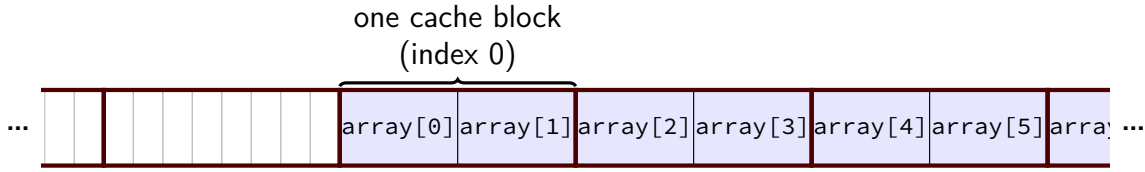
```
int array[8];
...
int even_sum = 0, odd_sum = 0;
even_sum += array[0];
odd_sum += array[1];
even_sum += array[2];
odd_sum += array[3];
even_sum += array[4];
odd_sum += array[5];
even_sum += array[6];
odd_sum += array[7];
```

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

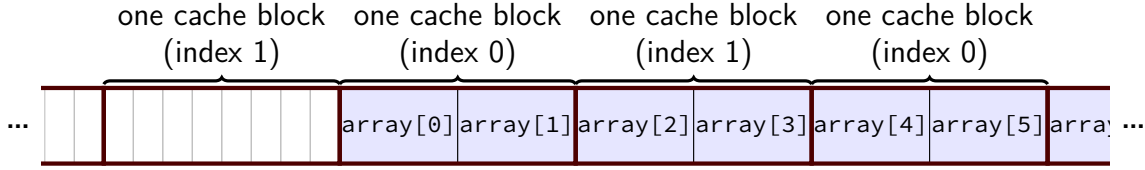
Assume array[0] at beginning of cache block.

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

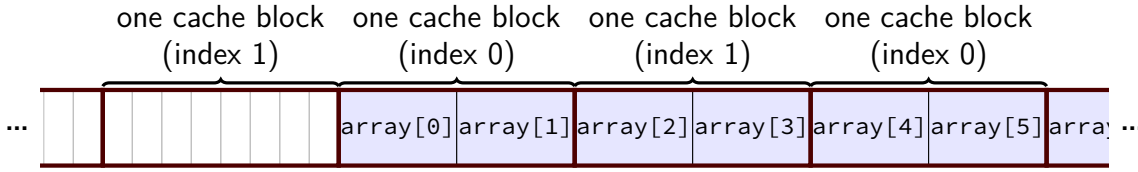
exercise solution



exercise solution

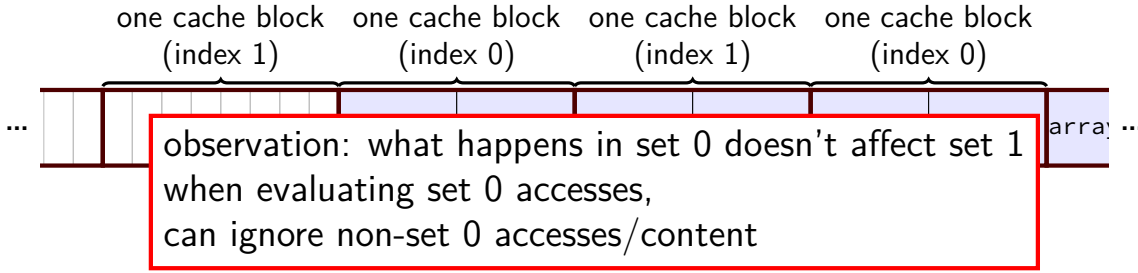


exercise solution



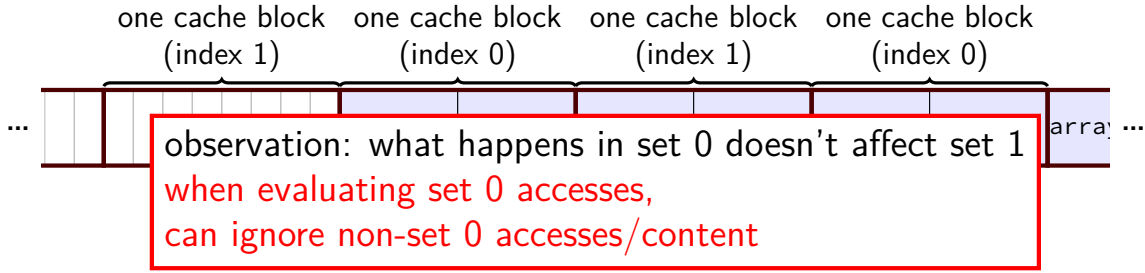
memory access	**set 0 afterwards**	**set 1 afterwards**
—	(empty)	(empty)
read array[0] (miss)	{array[0], array[1]}	(empty)
read array[1] (hit)	{array[0], array[1]}	(empty)
read array[2] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[3] (hit)	{array[0], array[1]}	{array[2], array[3]}
read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

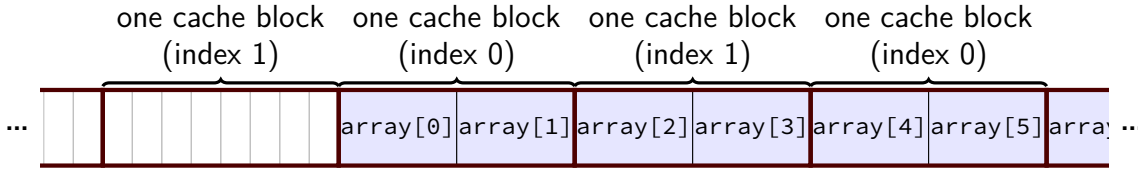


observation: what happens in set 0 doesn't affect set 1

when evaluating set 0 accesses,
can ignore non-set 0 accesses/content

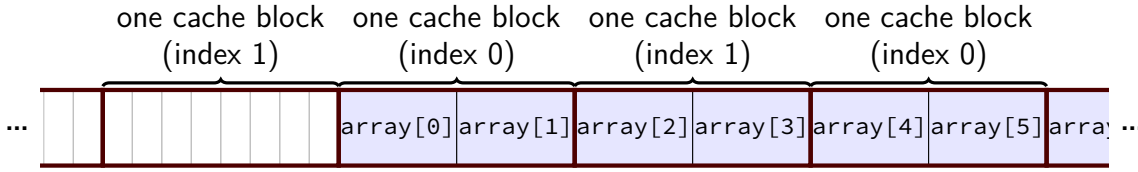
memory access	set 0 afterwards	set 1 afterwards
—	(empty)	(empty)
read array[0] (miss)	{array[0], array[1]}	(empty)
read array[1] (hit)	{array[0], array[1]}	(empty)
read array[2] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[3] (hit)	{array[0], array[1]}	{array[2], array[3]}
read array[4]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

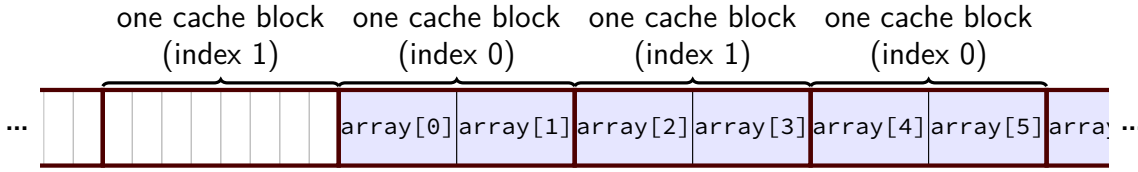
C and cache misses (warmup 4)

```
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).

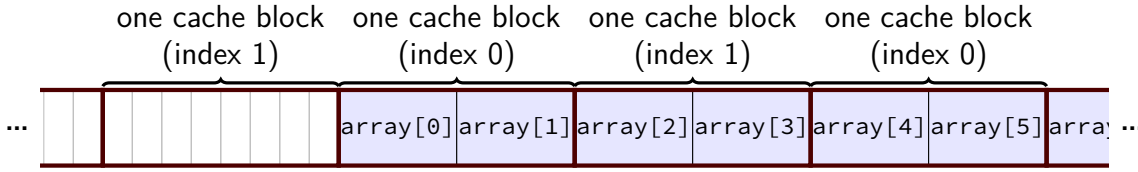
How many data cache misses on a **2**-set direct-mapped cache with 8B blocks?

exercise solution



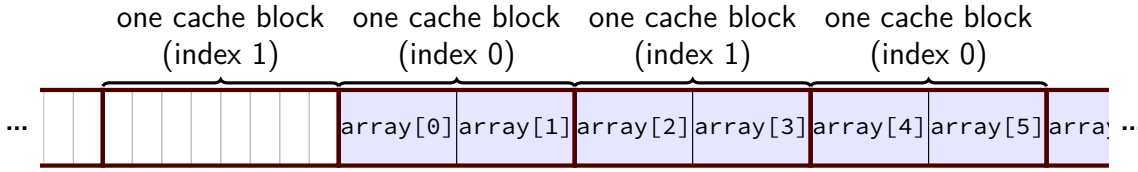
memory access	**set 0 afterwards**	**set 1 afterwards**
—	(empty)	(empty)
read array[0] (miss)	{array[0], array[1]}	(empty)
read array[2] (miss)	{array[0], array[1]}	{array[2], array[3]}
read array[4] (miss)	{array[4], array[5]}	{array[2], array[3]}
read array[6] (miss)	{array[4], array[5]}	{array[6],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

backup slides

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 not increase AMAT?

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 not increase AMAT?

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 not increase AMAT?

miss rate of $2/30 \rightarrow$ approx 93% hit rate