cache performance 1

changelog

25 Oct 2022: cache tradeoffs: change ‘increase block size’ effect on hit time to ‘?’ instead of ‘bad’

26 Oct 2022: undo that change (after looking up paper on the subject)

last time

replacement policies

- when set is full, what to replace?
- assume locality? least recently used or approx.
- simpler? first-in, first-out; random

write policies: on a write should cache

- store new block? *write-allocate* (v. write-no-allocate)
- update next level immediately? *write-through* (v. write-back)

average memory access time (start)

- hit time + miss rate · miss penalty
- miss penalty = miss time - hit time
  - ≈ extra time after checking for + not having hit

quiz Q1

tag 0xC20: 1100 0010 0000

set index 1 (4 sets total = 2 set index bits): 01

2nd byte in block – offset (8 bytes in block = 3 block offset bits): 001

1 1000 0100 0000 1001 = 0x18409

quiz Q2

value being replaced — goes in full set

if set 0: [TAG]00[OFFSET]

- offset: any 3 bits
- tag: anything but 0xA20 and 0xC20

if set 2: [TAG]10[OFFSET]

- offset: any 3 bits
- tag: anything but 0xC20 and 0x332

quiz Q3 [writes]

writes don't affect what's in cache + always go to next level

- 40 writes total go to next level

quiz Q3 [mapping]

2-byte blocks → cache blocks start at even addresses

if A[0] is at even address:

    `{A[0], A[1]}` are together in a block (in set 0/1)
    `{A[2], A[3]}` are together in a block (in set 1/0)

if A[0] is at odd address:

    `{*A[-1], A[0]}` are together in a block (in set 0/1)
    `{A[1], A[2]}` are together in a block (in set 1/0)
    `{A[3], *A[4]}` are together in a block (in set 0/1)

will talk more about this issue later today

quiz Q3 [reads, even]

if A start at multiple of 4 address, 2 reads total:

	set 0	set 1	
	(empty)	empty	
read A[0]	A[0],A[1]	—	+1 read next level
read A[1]	A[0],A[1]	—	
read A[2]	A[0],A[1]	A[2],A[3]	+1 read next level
read A[3]	A[0],A[1]	A[2],A[3]	
...	...	...	
read A[0]	A[0],A[1]	A[2],A[3]	
read A[1]	A[0],A[1]	A[2],A[3]	
read A[2]	A[0],A[1]	A[2],A[3]	
read A[3]	A[0],A[1]	A[2],A[3]	
...	...	...	

quiz Q3 [reads, odd]

if A start at odd address, $3 + 2 \cdot 9 = 21$ reads

	set 0	set 1	
	(empty)	(empty)	
read A[0]	A[-1],A[0]	—	+1 read
read A[1]	A[-1],A[0]	A[1],A[2]	+1 read
read A[2]	A[-1],A[0]	A[1],A[2]	
read A[3]	A[3],A[4]	A[1],A[2]	+1 read
...	...	...	
read A[0]	A[-1],A[0]	A[1],A[2]	+1 read
read A[1]	A[-1],A[0]	A[1],A[2]	
read A[2]	A[-1],A[0]	A[1],A[2]	
read A[3]	A[3],A[4]	A[1],A[2]	+1 read
...	...	...	

quiz Q3 overall

A, B both at even addresses

40 (writes) + 2 (read A) + 2 (read B) = 44

A at odd address, B at even:

40 (writes) + 21 (read A) + 2 (read B) = 63

A at even address, B at odd:

40 (writes) + 2 (read A) + 3 (read B) = 45

A, B both at odd addresses

40 (writes) + 20 (read A) + 3 (read B) = 64

quiz Q4

4MB (data size) $= 2^{22}$ byte

64 byte blocks $\rightarrow$ 6 block offset bits

$2^{22} \div (4 \cdot 64) = 2^{14}$ sets $\rightarrow$ 14 set index bits

64 - (14 $+$ 6) $=$ 44 tag bits per block

2^{14} (sets) $\cdot 4$ (blocks/set) $= 2^{16}$ blocks

$44 \cdot 2^{16} = 2\,883\,584$ bits of tags stored

quiz Q5-6 (pt 1)

write 0x4444000	set 0x1100, tag 0x44	+valid, +dirty (valid b/c write-allocate) (dirty b/c write-back)
write 0x4444008	set 0x1100, tag 0x44	already in cache
read 0x4444000	set 0x1100, tag 0x44	already in cache
read 0x4444800	set 0x1120, tag 0x44	+valid
read 0x4444320	set 0x10cc, tag 0x44	+valid
write 0x4445000	set 0x1140, tag 0x44	+valid (write-allocate), +dirty (write-back)
read 0x8445000	set 0x1140, tag 0x84	+valid (same set as before but only using 2 of 4 ways)

5 valid bits set

2 dirty bits set

average memory access time

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

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

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 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?

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

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

cache miss types

common to categorize misses:

- roughly “cause” of miss assuming cache block size fixed

compulsory (or *cold*) — first time accessing something

- adding more sets or blocks/set wouldn’t change

conflict — sets aren’t big/flexible enough

- a fully-associtive (1-set) cache of the same size would have done better

capacity — cache was not big enough

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

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 + keeping cache size constant if possible...)

	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 organization and miss rate

depends on program; one example:

SPEC CPU2000 benchmarks, 64B block size

LRU replacement policies

data cache miss rates:

Cache size	direct-mapped	2-way	8-way	fully assoc.
1KB	8.63%	6.97%	5.63%	5.34%
2KB	5.71%	4.23%	3.30%	3.05%
4KB	3.70%	2.60%	2.03%	1.90%
16KB	1.59%	0.86%	0.56%	0.50%
64KB	0.66%	0.37%	0.10%	0.001%
128KB	0.27%	0.001%	0.0006%	0.0006%

Data: Cantin and Hill, "Cache Performance for SPEC CPU2000 Benchmarks"
`http://research.cs.wisc.edu/multifacet/misc/spec2000cache-data/`

cache organization and miss rate

depends on program; one example:

SPEC CPU2000 benchmarks, 64B block size

LRU replacement policies

data cache miss rates:

Cache size	direct-mapped	2-way	8-way	fully assoc.
1KB	8.63%	6.97%	5.63%	5.34%
2KB	5.71%	4.23%	3.30%	3.05%
4KB	3.70%	2.60%	2.03%	1.90%
16KB	1.59%	0.86%	0.56%	0.50%
64KB	0.66%	0.37%	0.10%	0.001%
128KB	0.27%	0.001%	0.0006%	0.0006%

Data: Cantin and Hill, "Cache Performance for SPEC CPU2000 Benchmarks"
`http://research.cs.wisc.edu/multifacet/misc/spec2000cache-data/`

cache accesses and C code (1)

```c
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 `0x7fffffffe43b8`
- scaleFactor located at address `0x6bc3a0`

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

	return address	scaleFactor
tag	0xfffffffc	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	0xfffffffc	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

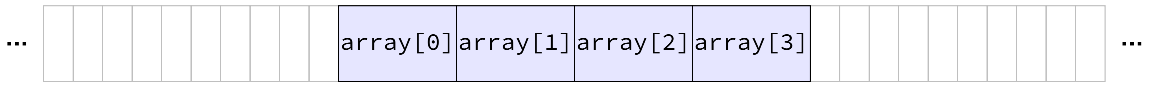
C and cache misses (warmup 1)

```
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).

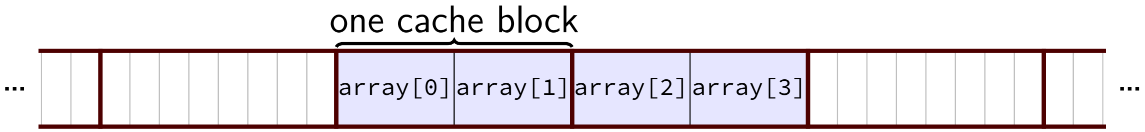
How many data cache misses on a 1-set direct-mapped cache with 8B blocks?

some possiblities



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

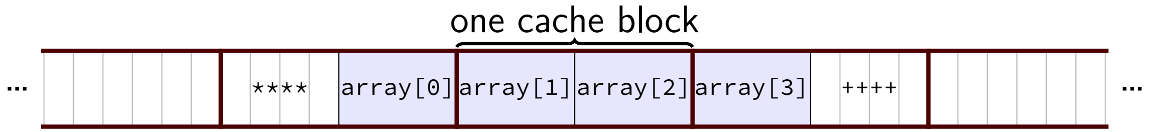
some possiblities



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]}`

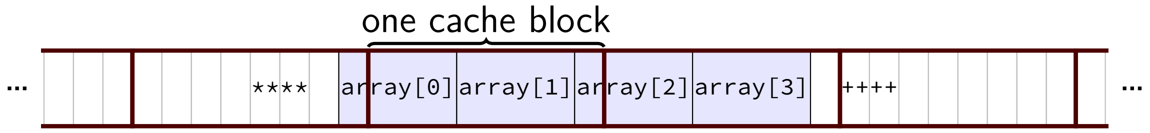
some possiblities



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)

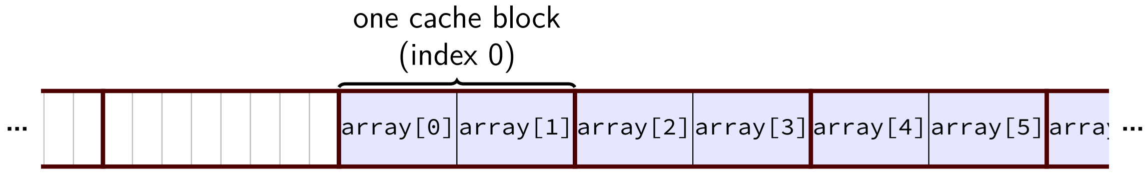
```
int array[8];
...
int even_sum = 0, odd_sum = 0;
even_sum += array[0];
odd_sum += array[1];
even_sum += array[2];
odd_sum += array[3];
even_sum += array[4];
odd_sum += array[5];
even_sum += array[6];
odd_sum += array[7];
```

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

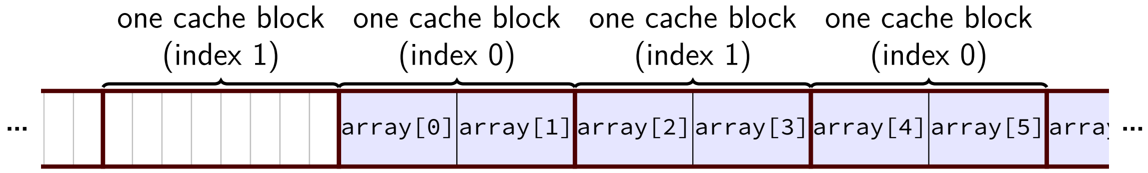
Assume array[0] at beginning of cache block.

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

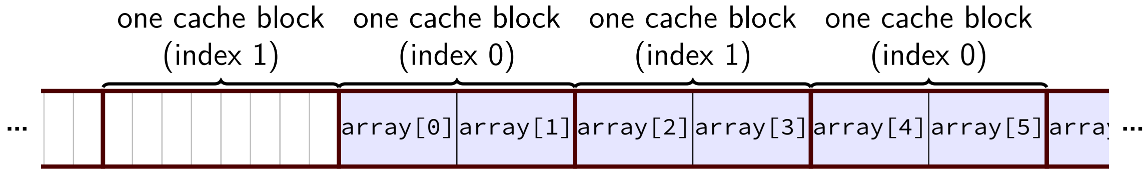
exercise solution



exercise solution


one cache block (index 1)
one cache block (index 0)
one cache block (index 1)
one cache block (index 0)
...
array[0]
array[1]
array[2]
array[3]
array[4]
array[5]
arra
...

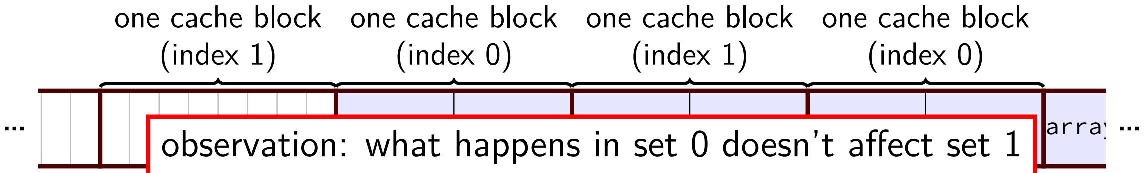
exercise solution



one cache block (index 1)
one cache block (index 0)
one cache block (index 1)
one cache block (index 0)
...
array[0]
array[1]
array[2]
array[3]
array[4]
array[5]
arra
...

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]}

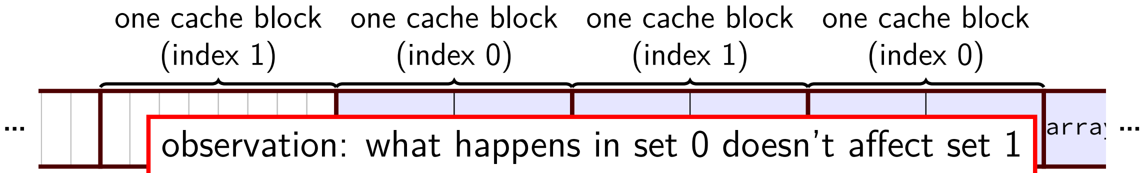
exercise solution


one cache block (index 1)
one cache block (index 0)
one cache block (index 1)
one cache block (index 0)
...
array
...

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

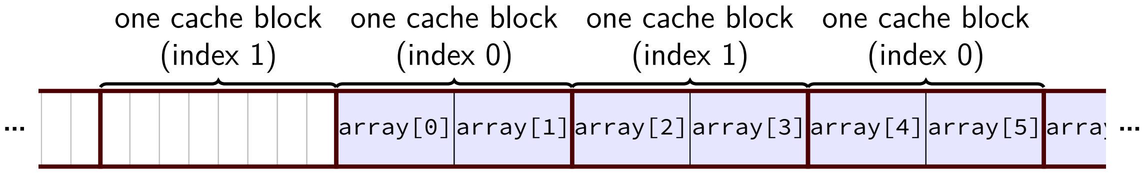


one cache block (index 1)
one cache block (index 0)
one cache block (index 1)
one cache block (index 0)
...
array
...

observation: what happens in set 0 doesn't affect set 1
when evaluating set 0 accesses,
can ignore non-set 0 accesses/content

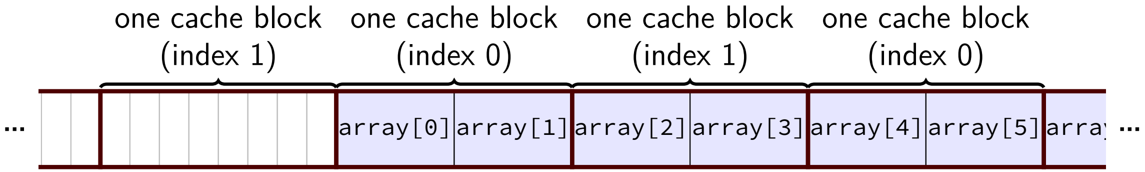
memory access	set 0 afterwards	set 1 afterwards
—	(empty)	(empty)
read `array[0]` (miss)	{array[0], array[1]}	(empty)
read `array[1]` (hit)	{array[0], array[1]}	(empty)
read `array[2]` (miss)	{array[0], array[1]}	{array[2], array[3]}
read `array[3]` (hit)	{array[0], array[1]}	{array[2], array[3]}
read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


one cache block (index 1)
one cache block (index 0)
one cache block (index 1)
one cache block (index 0)
...
array[0]
array[1]
array[2]
array[3]
array[4]
array[5]
array
...

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


one cache block (index 1)
one cache block (index 0)
one cache block (index 1)
one cache block (index 0)
...
array[0]
array[1]
array[2]
array[3]
array[4]
array[5]
array
...

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]}

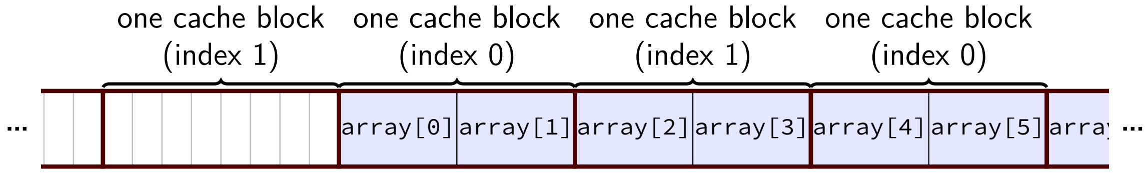
C and cache misses (warmup 4)

```
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).

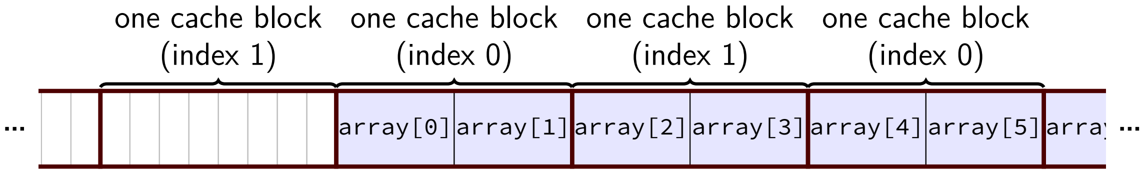
How many data cache misses on a **2**-set direct-mapped cache with 8B blocks?

exercise solution


one cache block (index 1)
one cache block (index 0)
one cache block (index 1)
one cache block (index 0)
...
array[0]
array[1]
array[2]
array[3]
array[4]
array[5]
array
...

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]}

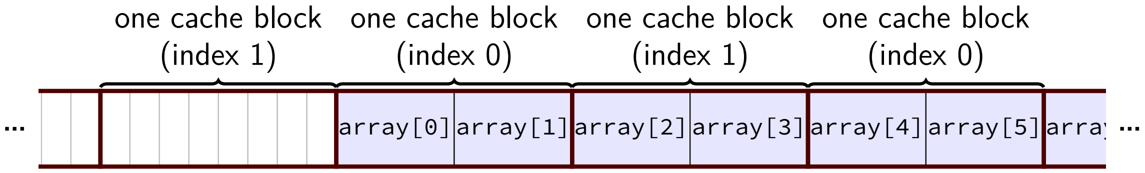
exercise solution



one cache block (index 1)
one cache block (index 0)
one cache block (index 1)
one cache block (index 0)
...
array[0]
array[1]
array[2]
array[3]
array[4]
array[5]
array
...

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



one cache block (index 1)
one cache block (index 0)
one cache block (index 1)
one cache block (index 0)
...
array[0]
array[1]
array[2]
array[3]
array[4]
array[5]
array
...

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-associtiave cache be better?

backup slides

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

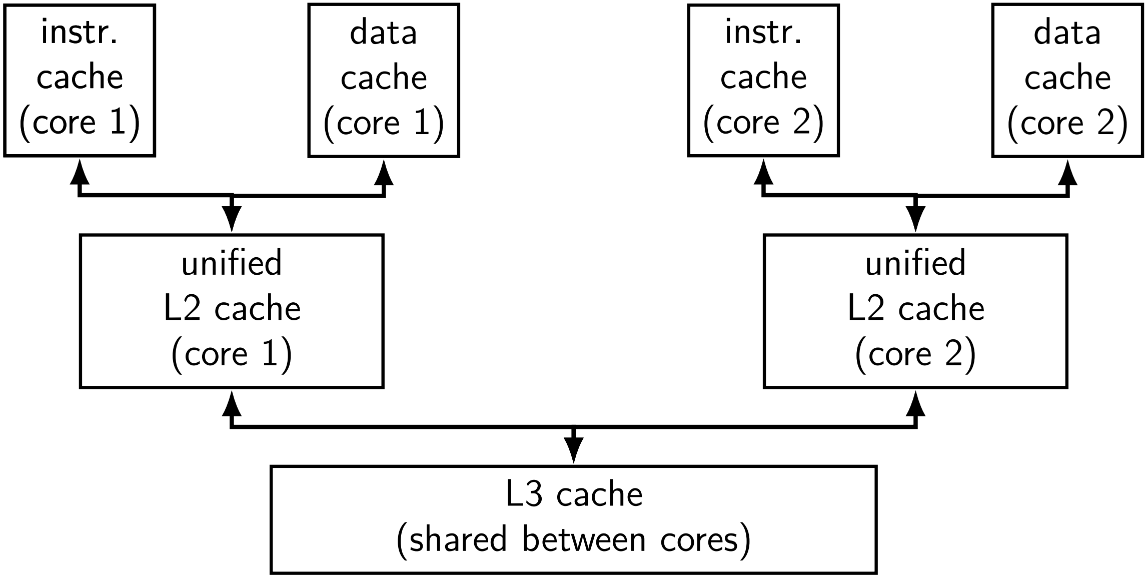
exercise (3)

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

split caches; multiple cores


instr. cache (core 1)
data cache (core 1)
instr. cache (core 2)
data cache (core 2)
unified L2 cache (core 1)
unified L2 cache (core 2)
L3 cache (shared between 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

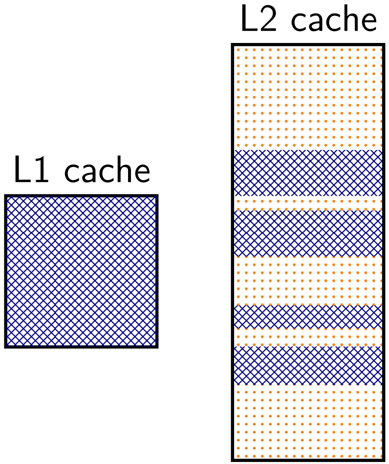
can optimize instruction cache for different access pattern

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

inclusive versus exclusive

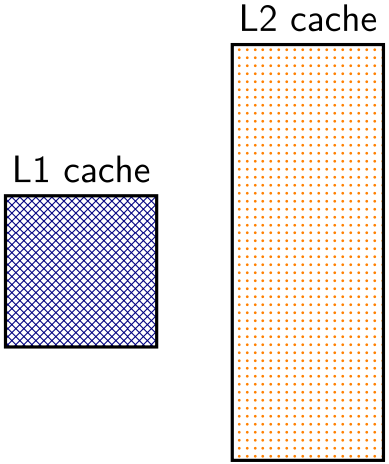
L2 inclusive of L1

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



L2 exclusive of L1

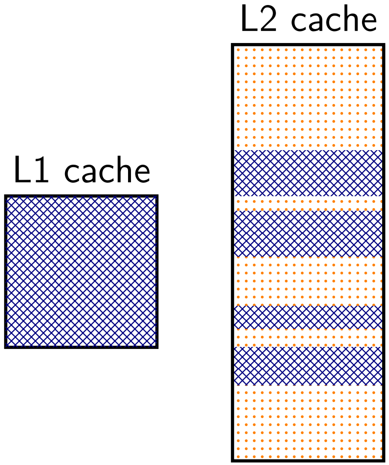
L2 contains different data than L1
adding to L1 must remove from L2
probably evicting from L1 adds to L2



inclusive versus exclusive

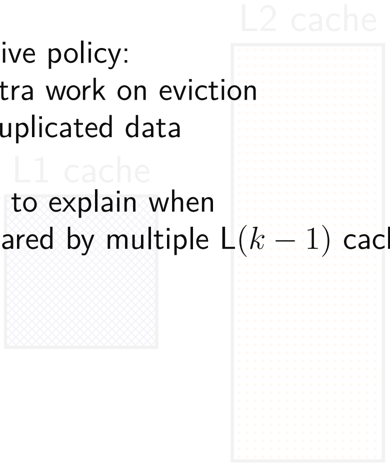
L2 inclusive of L1

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



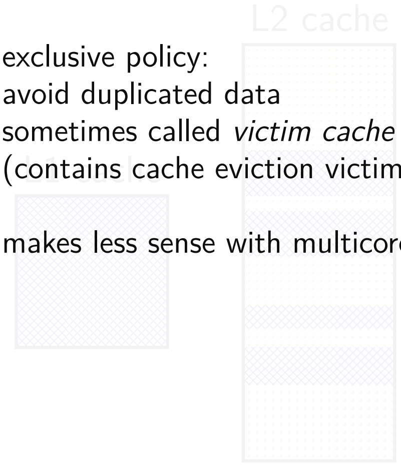
inclusive policy:
no extra work on eviction
but duplicated data

easier to explain when
Lk 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

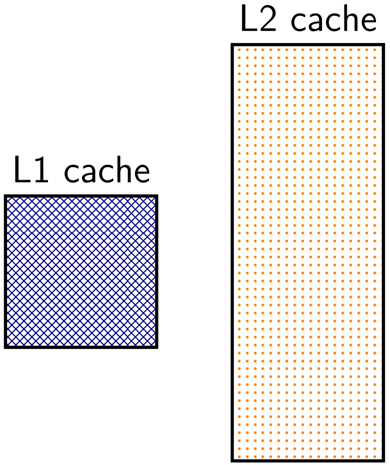


exclusive policy:
avoid duplicated data
sometimes called *victim cache*
(contains cache eviction victims)

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


L2 cache
L1 cache