last time (1)

- locality — temporal and spatial
  - temporal: same thing again soon
  - spatial: nearby thing soon
  - natural properties of programs
  - some taken advantage of by compiler (register allocation)

- direct-mapped caches
  - divide memory, cache into blocks
  - always power-of-two size blocks, number of ‘rows’ in cache
  - one place to put each block of memory in the cache

last time (2)

direct-mapped cache lookup

- divide address into tag / (set) index / (block) offset
- b-bit block offset — where in 2^b block is byte?
- s-bit set index — which of 2^s rows of cache to use?
- tag — which block from memory is stored here?
- (could store whole block address instead of tag, just saving space)

instruction v data caches

alignment and C code

- want to avoid splitting things across blocks
- better start at beginning of block ($=$ multiple of block size)

anonymous feedback

“The quiz this week was absurdly ambiguous and difficult to understand.”

“The quiz for this week was incredibly ambiguous for questions 4 and 5. In DSA, we learned about two ways to resolve collisions in hash tables (separate chaining and probing). Depending on which method is, the locality answers in those questions will be greatly effected.”

quiz Q1 (1)

"contains a public encryption key used by the web browser to encrypt, among other things, the path of the web page being requested"

TLS protocol client and server agree on symmetric keys

- use key in certificate here!
- most commonly: to sign one-time key share for server

then use symmetric keys to encrypt rest of connection

why?

- symmetric encryption faster
- forward secrecy — server can't decrypt old connections retrospectively

quiz Q1 (2)

“is verified primarily web browser contacting certificate authority”

certificate contains signature that can be checked with just CA public key

avoids scaling problem of every browser contacting CA every time

(yes, is true that revocation information might contact CA,
but not required/often ignored if CA not available and/or done indirectly)

quiz Q2

A→B: A's key share (from secret S_A)

B→A: B's key share (from secret S_B)

"using the key shares sent by A and B as well as their own secret value and key share, compute their own copy of the symmetric encryption key A and B are using"

- can't compute A and B's secret from their key shares alone
- typical attack: replace one of the key shares

scenario that 'works': replace A's key share with attacker's for B and replace B's key share with attacerk's for A

quiz Q4

hashtable:

- spatial: not really, spread out (maybe except *rare* collisions)
- temporal: yes, for duplicated eliminates

input array:

- spatial: yes, iterating through sequentially most likely
- temporal: no, each element used essentially once (maybe once to hash, once to equality compare — still not much)

quiz Q5

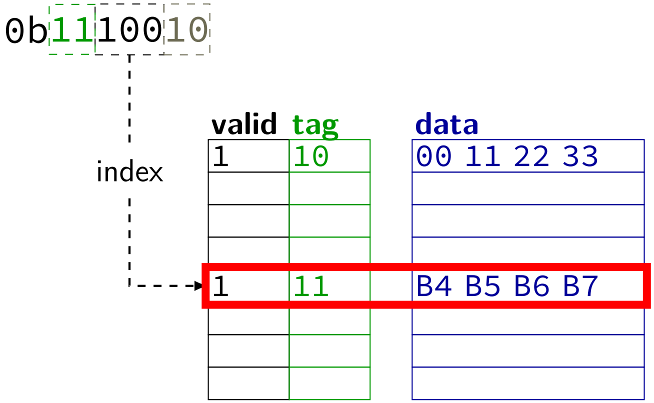
hashtable with better hash function (less collisions)

improves locality (either kind) — no: better spread

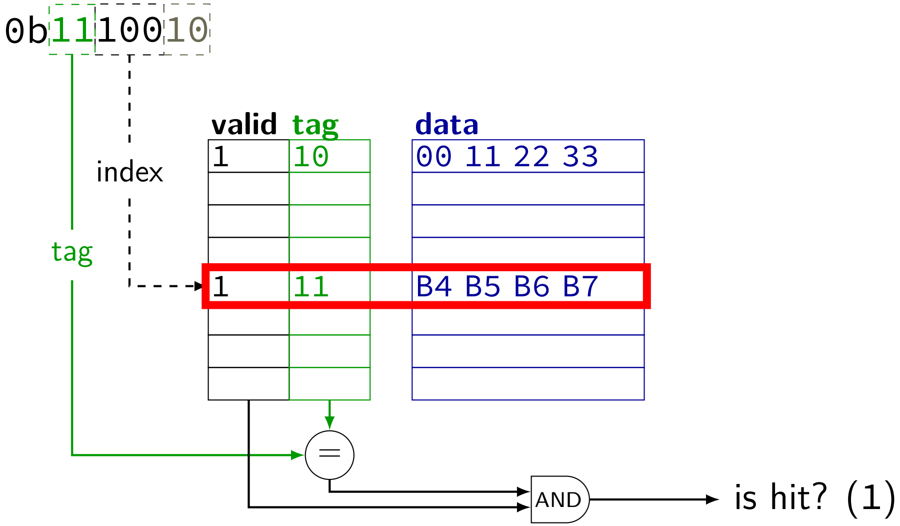
- less spatial if probing sequentially
- less traversing lists/proving entries also traversed for other values

reduces accesses — yes, less traversing lists/probing for collisions

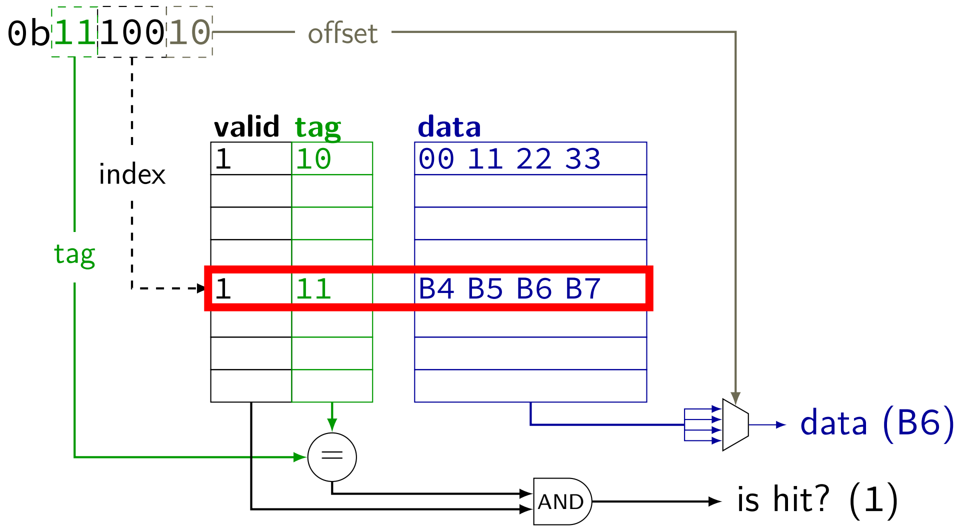
cache operation (read)



cache operation (read)



cache operation (read)



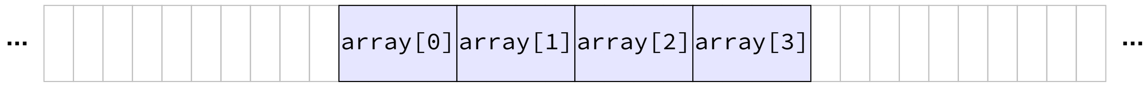
C and cache misses (warmup 1)

```
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).

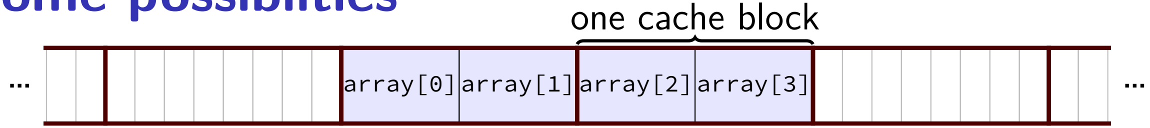
How many data cache misses on a 1-set direct-mapped cache with 8B blocks?

some possiblities



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

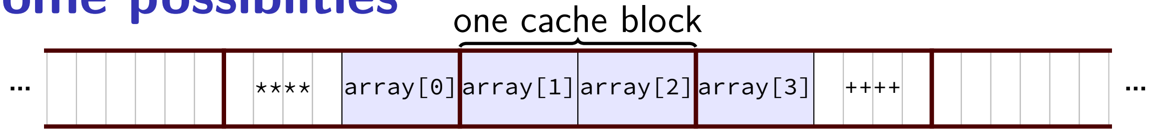
some possiblities



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]}`

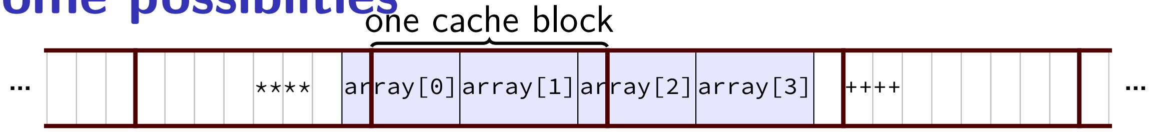
some possiblities



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)

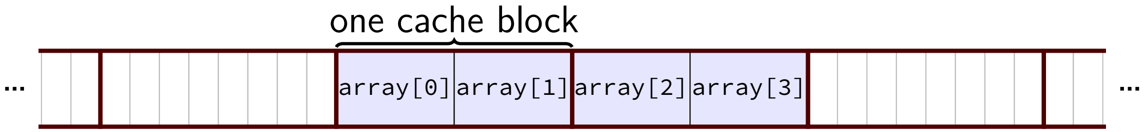
```
int array[4];
int even_sum = 0, odd_sum = 0;
even_sum += array[0];
even_sum += array[2];
odd_sum += array[1];
odd_sum += array[3];
```

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?

exercise solution



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

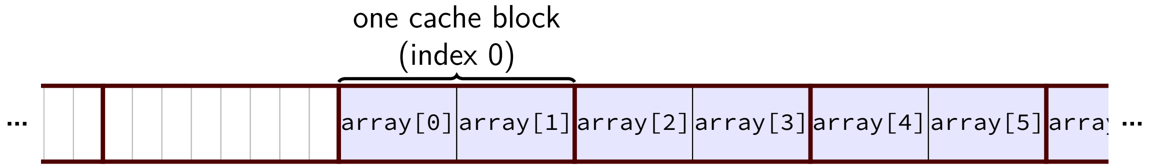
C and cache misses (warmup 3)

```
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];
```

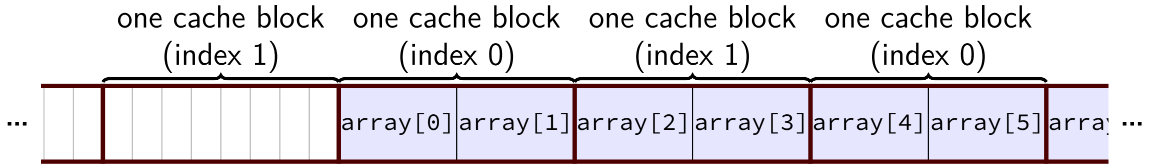
Assume everything but `array` is kept in registers (and the compiler does not do anything funny), and array[0] at beginning of cache block.

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

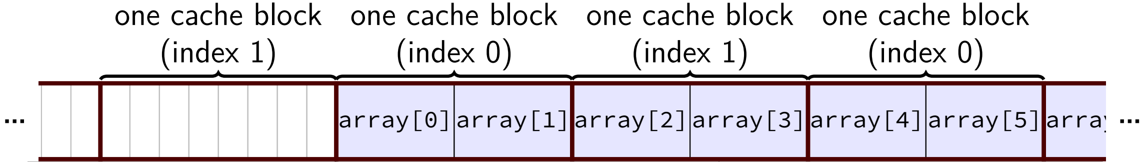
exercise solution



exercise solution

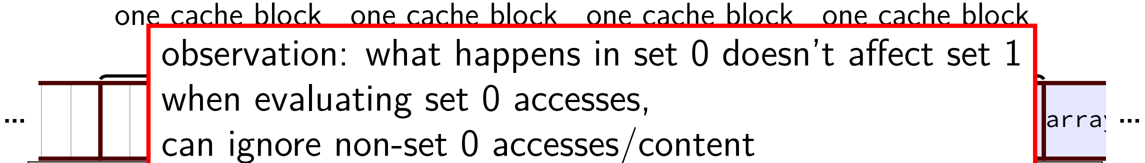

one cache block (index 1)
one cache block (index 0)
one cache block (index 1)
one cache block (index 0)
...
array[0]
array[1]
array[2]
array[3]
array[4]
array[5]
arra
...

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]}

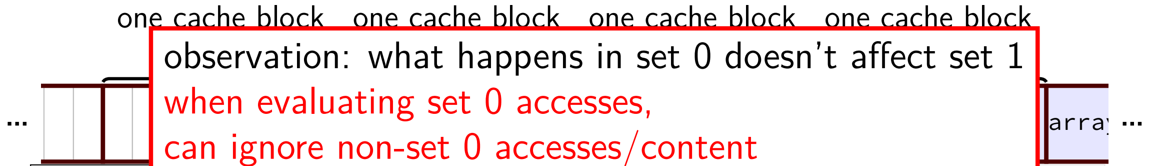
exercise solution



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

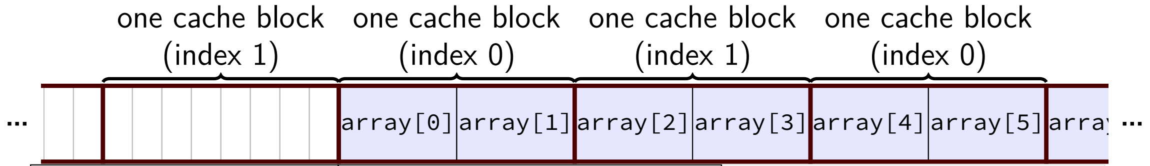


observation: what happens in set 0 doesn't affect set 1

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

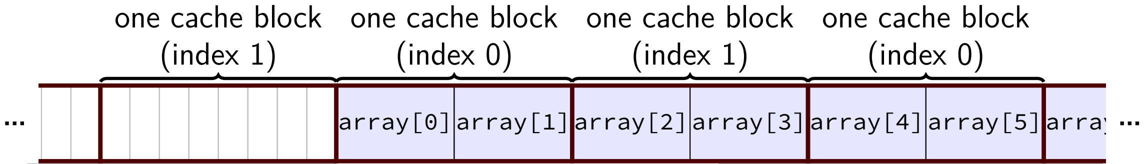
memory access	set 0 afterwards	set 1 afterwards
—	(empty)	(empty)
read `array[0]` (miss)	`{array[0], array[1]}`	(empty)
read `array[1]` (hit)	`{array[0], array[1]}`	(empty)
read `array[2]` (miss)	`{array[0], array[1]}`	`{array[2], array[3]}`
read `array[3]` (hit)	`{array[0], array[1]}`	`{array[2], array[3]}`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

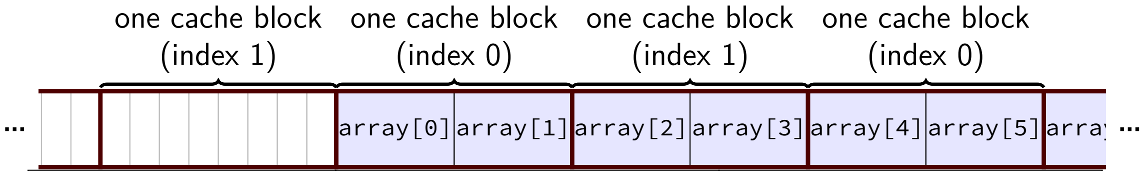
C and cache misses (warmup 4)

```
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).

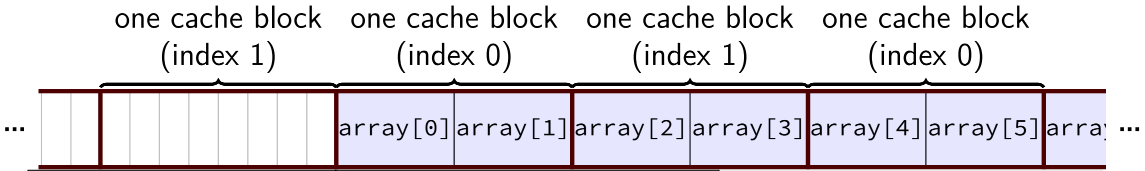
How many data cache misses on a **2**-set direct-mapped cache with 8B blocks?

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]
arra
...

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]}

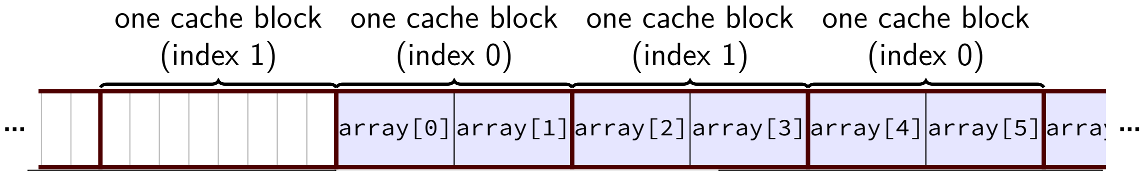
exercise solution



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]
arra
...

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]
arra
...

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]}

cache size

cache size = amount of *data* in cache

not included metadata (tags, valid bits, etc.)

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 initially empty 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 initially empty 2KB direct-mapped cache with 16B cache blocks?

arrays and cache misses (2b)

```
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 initially empty 4KB direct-mapped cache with 16B cache blocks?

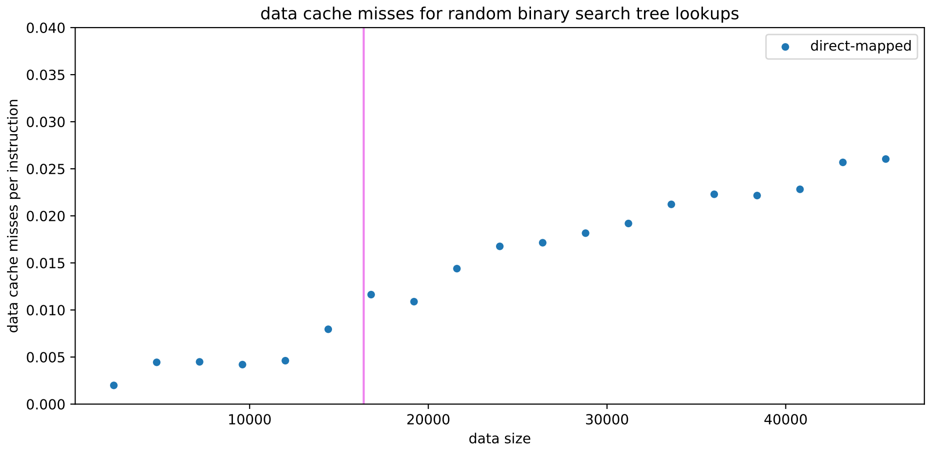
arrays and cache misses (3)

```
int sum; int array[1024]; // 4KB array
for (int i = 8; i < 1016; i += 1) {
    int local_sum = 0;
    for (int j = i - 8; j < i + 8; j += 1) {
        local_sum += array[i] * (j - i);
    }
    sum += (local_sum - array[i]);
}
```

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

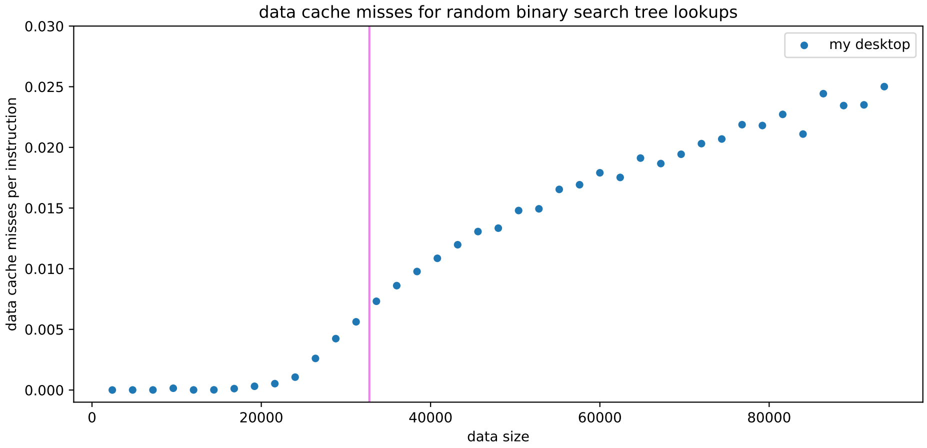
How many *data cache misses* on initially empty 2KB direct-mapped cache with 16B cache blocks?

simulated misses: BST lookups



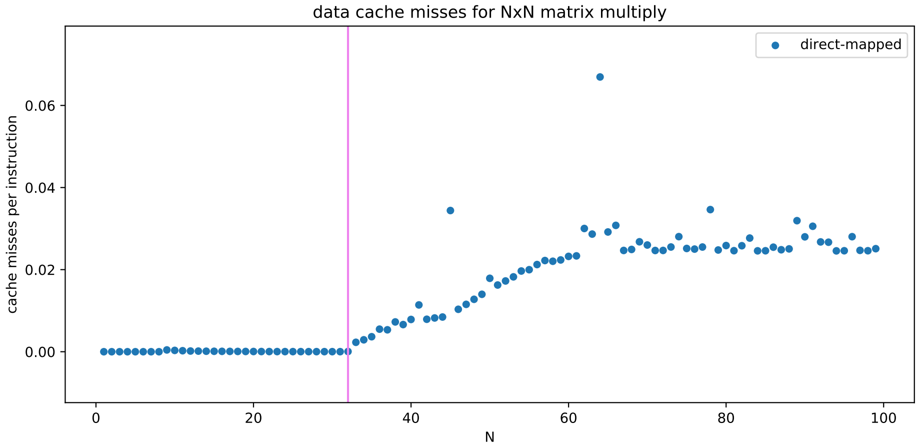
(simulated 16KB direct-mapped data cache; excluding BST setup)

actual misses: BST lookups



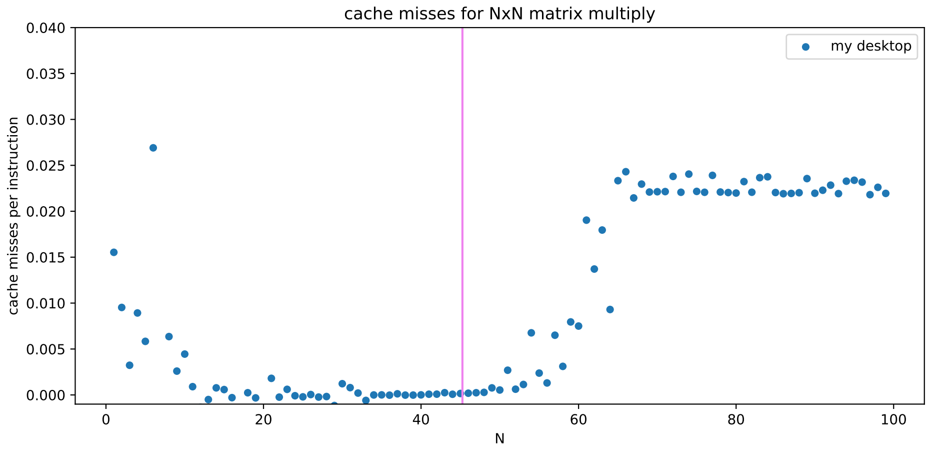
(actual 32KB more complex data cache)
(only one set of measurements + other things on machine + excluding initial load)

simulated misses: matrix multiplies



(simulated 16KB direct-mapped data cache; excluding initial load)

actual misses: matrix multiplies



(actual 32KB more complex data cache; excluding matrix initial load)
(only one set of measurements + other things on machine)

misses with skipping

```
int array1[512]; int array2[512];
...
for (int i = 0; i < 512; i += 1)
    sum += array1[i] * array2[i];
}
```

Assume everything but `array1`, `array2` is kept in registers (and the compiler does not do anything funny).

About how many *data cache misses* on a 2KB direct-mapped cache with 16B cache blocks?
Hint: depends on relative placement of array1, array2

best/worst case

`array1[i]` and `array2[i]` always different sets:

- = distance from array1 to array2 not multiple of # sets $\times$ bytes/set
- 2 misses every 4 `i`
- blocks of 4 array1[X] values loaded, then used 4 times before loading next block
- (and same for array2[X])

`array1[i]` and `array2[i]` same sets:

- = distance from array1 to array2 is multiple of # sets $\times$ bytes/set
- 2 misses every `i`
- block of 4 array1[X] values loaded, one value used from it, then, block of 4 array2[X] values replaces it, one value used from it, …

worst case in practice?

two rows of matrix?

often sizeof(row) bytes apart

if the row size is multiple of number of sets $\times$ bytes per block, oops!

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	**valid**	**tag**	**value**	**valid**	**tag**	**value**
`0`	`0`			`0`		
`1`	`0`			`0`		

multiple places to put values with same index
avoid misses from two active values using same set
("conflict misses"))

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	0		set 0	0		
1	0		set 1	0		

adding associativity

2-way set associative, 2 byte blocks, 2 sets

	way 0			way 1		
index	**valid**	**tag**	**value**	**valid**	**tag**	**value**
`0`	`0`			`0`		
`1`	`0`			`0`		

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	0			0		
1	0			0		

$m = 8$ bit addresses
$S = 2 = 2^s$ sets
$s = 1$ (set) index bits

$B = 2 = 2^b$ byte block size
$b = 1$ (block) offset bits
$t = m - (s + b) = 6$ tag bits

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	0		
1	0			0		

address (hex)	result
00000000 (00)	miss
00000001 (01)	
01100011 (63)	
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	0		
1	0			0		

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	0		
1	1	011000	mem[0x62] mem[0x63]	0		

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	
01100010 (62)	
00000000 (00)	
01100100 (64)	

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]
1	1	011000	mem[0x62] mem[0x63]	0		

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	
00000000 (00)	
01100100 (64)	

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]
1	1	011000	mem[0x62] mem[0x63]	0		

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	
01100100 (64)	

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]
1	1	011000	mem[0x62] mem[0x63]	0		

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	hit
01100100 (64)	

adding associativity

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]
1	1	011000	mem[0x62] mem[0x63]	0		

address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	hit
01100100 (64)	miss

needs to replace block in set 0!

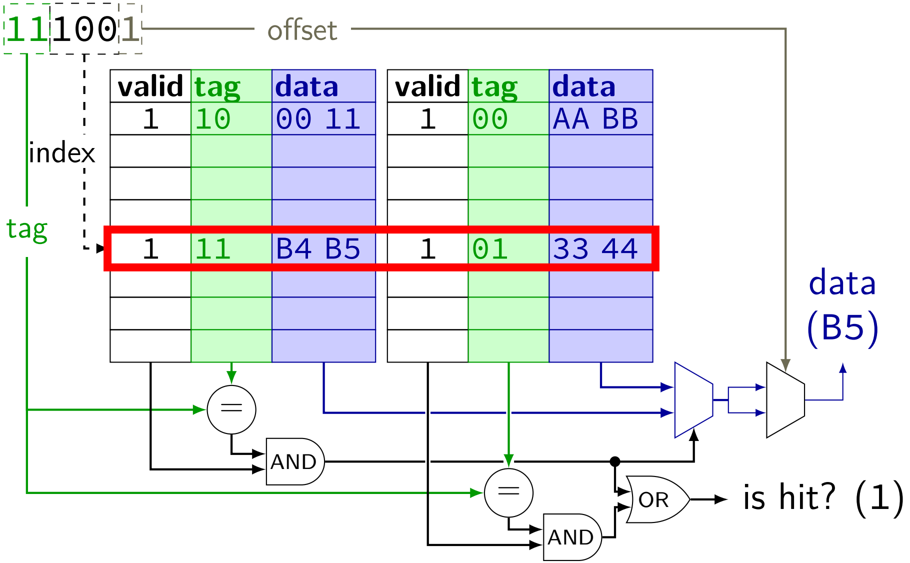
adding associativity

2-way set associative, 2 byte blocks, 2 sets

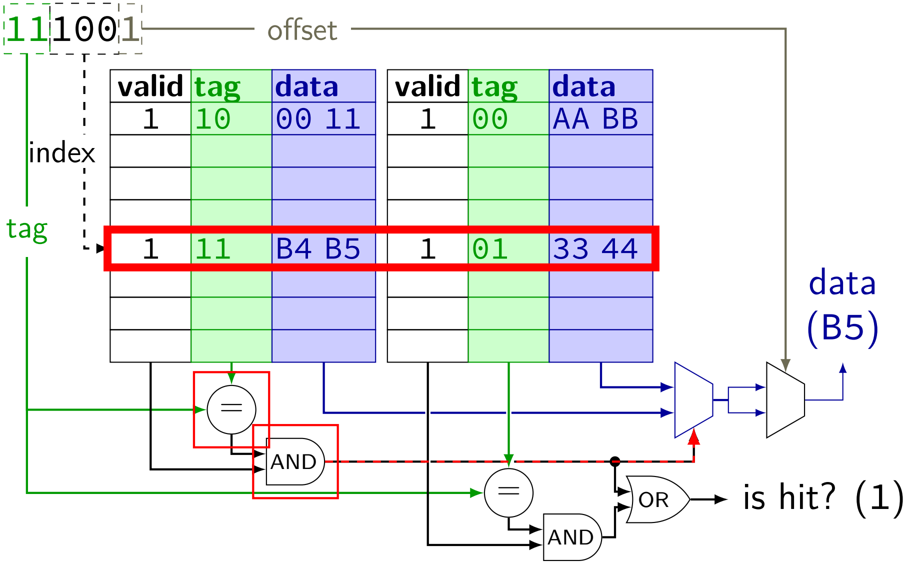
index	valid	tag	value	valid	tag	value
0	1	`000000`	`mem[0x00]` `mem[0x01]`	1	`011000`	`mem[0x60]` `mem[0x61]`
1	1	`011000`	`mem[0x62]` `mem[0x63]`	0		

address (hex)	result
`00000000 (00)`	miss
`00000001 (01)`	hit
`01100011 (63)`	miss
`01100001 (61)`	miss
`01100010 (62)`	hit
`00000000 (00)`	hit
`01100100 (64)`	miss

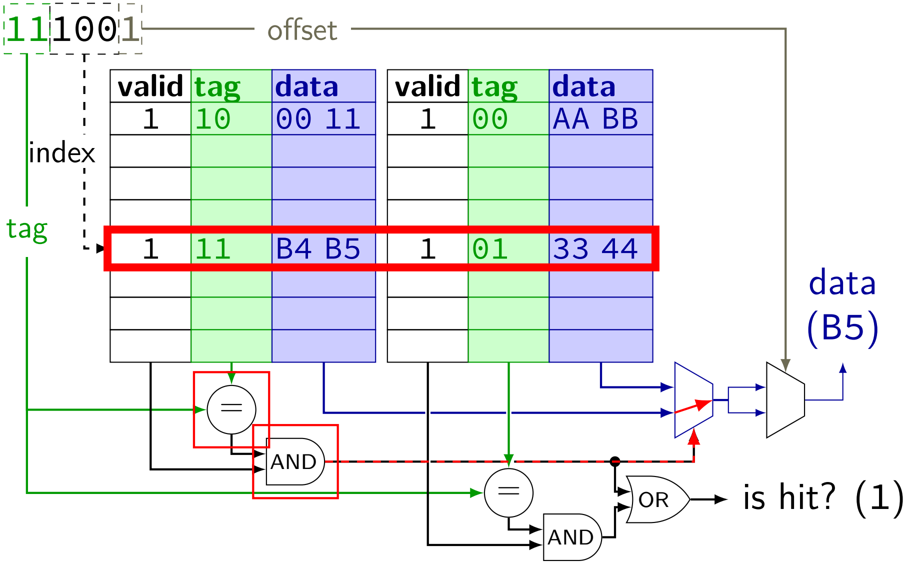
cache operation (associative)



cache operation (associative)



cache operation (associative)



associative lookup possibilities

none of the blocks for the index are valid

none of the valid blocks for the index match the tag

- something else is stored there

one of the blocks for the index is valid and matches the tag

replacement policies

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]
1	1	011000	mem[0x62] mem[0x63]	0		

address (hex)	result
000	
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	hit
01100100 (64)	miss

how to decide where to insert 0x64?

replacement policies

2-way set associative, 2 byte blocks, 2 sets

index	valid	tag	value	valid	tag	value	LRU
0	1	000000	mem[0x00] mem[0x01]	1	011000	mem[0x60] mem[0x61]	1
1	1	011000	mem[0x62] mem[0x63]	0			1

address (hex)	result
00000000 (00)	mi
00000001 (01)	hit
01100011 (63)	mi
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	hit
01100100 (64)	miss

track which block was read least recently
updated on every access

example replacement policies

least recently used

- take advantage of temporal locality
- at least $\lceil \log_2(E!) \rceil$ bits per set for E-way cache
  - (need to store order of all blocks)

approximations of least recently used

- implementing least recently used is expensive
- really just need "avoid recently used" — much faster/simpler
- good approximations: E to $2E$ bits

first-in, first-out

- counter per set — where to replace next

(pseudo-)random

- no extra information!
- actually works pretty well in practice

associativity terminology

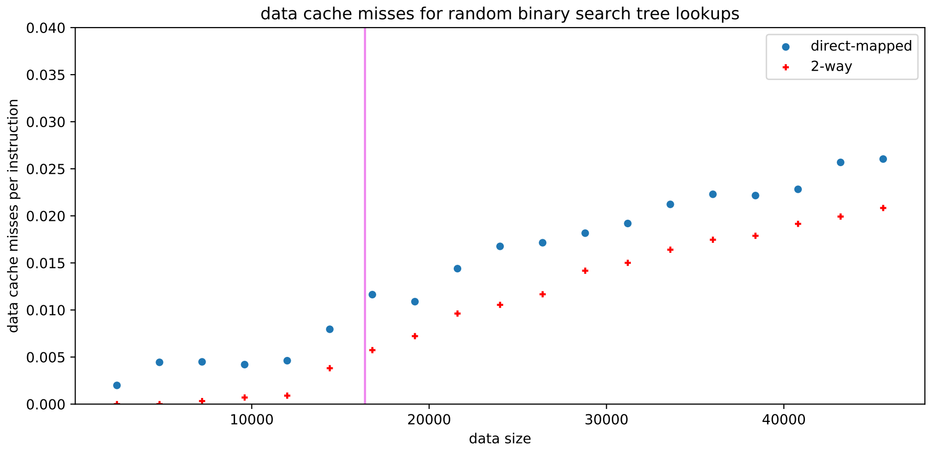
direct-mapped — one block per set

E-way set associative — E blocks per set

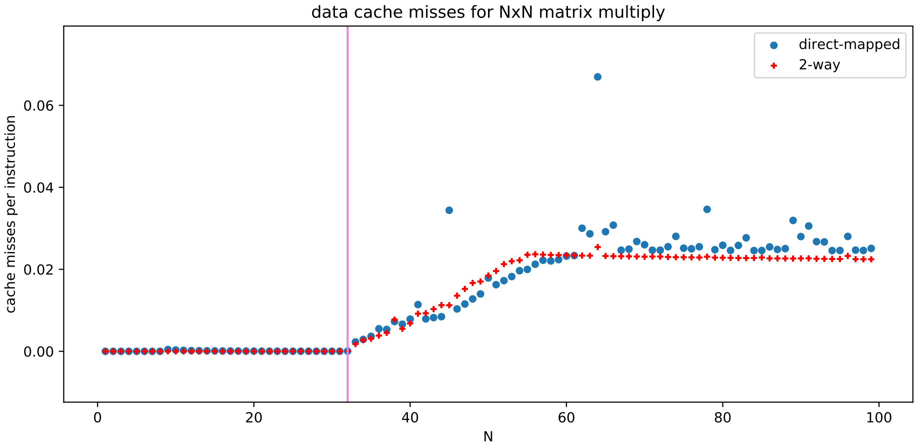
  E ways in the cache

fully associative — one set total (everything in one set)

simulated misses: BST lookups



simulated misses: matrix multiplies



logistics

Prof Skadron here Thursday

backup sides

Tag-Index-Offset formulas

m	memory addreses bits
E	number of blocks per set (“ways”)
$S = 2^s$	number of sets
s	(set) index bits
$B = 2^b$	block size
b	(block) offset bits
$t = m - (s + b)$	tag bits
$C = B \times S \times E$	cache size (excluding metadata)

Tag-Index-Offset exercise

m	memory addreses bits (Y86-64: 64)
E	number of blocks per set ("ways")
$S = 2^s$	number of sets
s	(set) index bits
$B = 2^b$	block size
b	(block) offset bits
$t = m - (s + b)$	tag bits
$C = B \times S \times E$	cache size (excluding metadata)

My desktop:

L1 Data Cache: 32 KB, 8 blocks/set, 64 byte blocks
L2 Cache: 256 KB, 4 blocks/set, 64 byte blocks
L3 Cache: 8 MB, 16 blocks/set, 64 byte blocks

Divide the address `0x34567` into tag, index, offset for each cache.

T-I-O exercise: L1

quantity	value for L1
block size (given)	$B = 64$Byte
	$B = 2^b$ (b: block offset bits)

T-I-O exercise: L1

quantity	value for L1
block size (given)	$B = 64$Byte
	$B = 2^b$ (b: block offset bits)
block offset bits	$b = 6$

T-I-O exercise: L1

quantity	value for L1
block size (given)	$B = 64$Byte
	$B = 2^b$ (b: block offset bits)
block offset bits	$b = 6$
blocks/set (given)	$E = 8$
cache size (given)	$C = 32$KB $= E \times B \times S$

T-I-O exercise: L1

quantity	value for L1
block size (given)	$B = 64\text{Byte}$
	$B = 2^b$ (b: block offset bits)
block offset bits	$b = 6$
blocks/set (given)	$E = 8$
cache size (given)	$C = 32\text{KB} = E \times B \times S$
	$S = \frac{C}{B \times E}$ (S: number of sets)

T-I-O exercise: L1

quantity	value for L1
block size (given)	$B = 64\text{Byte}$
	$B = 2^b$ (b: block offset bits)
block offset bits	$b = 6$
blocks/set (given)	$E = 8$
cache size (given)	$C = 32\text{KB} = E \times B \times S$
	$S = \frac{C}{B \times E}$ (S: number of sets)
number of sets	$S = \frac{32\text{KB}}{64\text{Byte} \times 8} = 64$

T-I-O exercise: L1

quantity	value for L1
block size (given)	$B = 64\text{Byte}$
block offset bits	$B = 2^b$ (b: block offset bits) $b = 6$
blocks/set (given)	$E = 8$
cache size (given)	$C = 32\text{KB} = E \times B \times S$
number of sets	$S = \frac{C}{B \times E}$ (S: number of sets) $S = \frac{32\text{KB}}{64\text{Byte} \times 8} = 64$
set index bits	$S = 2^s$ (s: set index bits) $s = \log_2(64) = 6$

T-I-O results

	L1	L2	L3
sets	64	1024	8192
block offset bits	6	6	6
set index bits	6	10	13
tag bits		(the rest)	

T-I-O: splitting

	L1	L2	L3
block offset bits	6	6	6
set index bits	6	10	13
tag bits	(the rest)		

```
           3    4    5    6    7
0x34567: 0011 0100 0101 0110 0111
```

bits 0-5 (all offsets): `100111` = `0x27`

T-I-O: splitting

	L1	L2	L3
block offset bits	6	6	6
set index bits	6	10	13
tag bits	(the rest)		

```
           3    4    5    6    7
0x34567: 0011 0100 0101 0110 0111
```

bits 0-5 (all offsets): `100111` = `0x27`

T-I-O: splitting

	L1	L2	L3
block offset bits	6	6	6
set index bits	6	10	13
tag bits		(the rest)	

```
          3    4    5    6    7
0x34567: 0011 0100 0101 0110 0111
```

bits 0-5 (all offsets): `100111` = `0x27`

L1:

bits 6-11 (L1 set): `01 0101` = `0x15`

bits 12- (L1 tag): `0x34`

T-I-O: splitting

	L1	L2	L3
block offset bits	6	6	6
set index bits	6	10	13
tag bits		(the rest)	

```
0x34567:   3    4    5    6    7
          0011 0100 0101 0110 0111
```

bits 0-5 (all offsets): `100111` = `0x27`

L1:

bits 6-11 (L1 set): `01 0101` = `0x15`

bits 12- (L1 tag): `0x34`

T-I-O: splitting

	L1	L2	L3
block offset bits	6	6	6
set index bits	6	10	13
tag bits		(the rest)	

```
          3    4    5    6    7
0x34567: 0011 0100 0101 0110 0111
```

bits 0-5 (all offsets): `100111` = `0x27`

L2:

bits 6-15 (set for L2): `01 0001 0101` = `0x115`

bits 16-: `0x3`

T-I-O: splitting

	L1	L2	L3
block offset bits	6	6	6
set index bits	6	10	13
tag bits		(the rest)	

```
           3    4    5    6    7
0x34567: 0011 0100 0101 0110 0111
```

bits 0-5 (all offsets): `100111` = `0x27`

L2:

bits 6-15 (set for L2): `01 0001 0101` = `0x115`
bits 16-: `0x3`

T-I-O: splitting

	L1	L2	L3
block offset bits	6	6	6
set index bits	6	10	13
tag bits		(the rest)	

```
               3      4      5      6      7
0x34567:    0011   0100   0101   0110   0111
```

bits 0-5 (all offsets): `100111` = `0x27`

L3:

bits 6-18 (set for L3): `0 1101 0001 0101` = `0xD15`

bits 18-: `0x0`

misses with skipping

```
int array1[512]; int array2[512];
...
for (int i = 0; i < 512; i += 1)
    sum += array1[i] * array2[i];
}
```

Assume everything but `array1`, `array2` is kept in registers (and the compiler does not do anything funny).

About how many *data cache misses* on a 2KB direct-mapped cache with 16B cache blocks?
Hint: depends on relative placement of array1, array2

How about on a two-way set associative cache?

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];
```

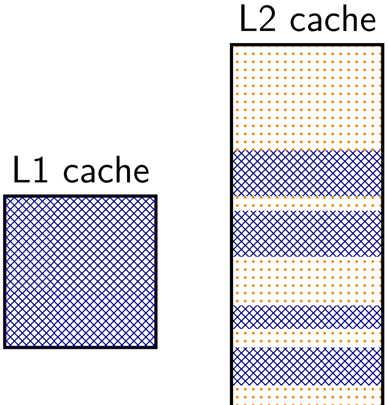
Assume everything but `array` is kept in registers (and the compiler does not do anything funny).

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

inclusive versus exclusive

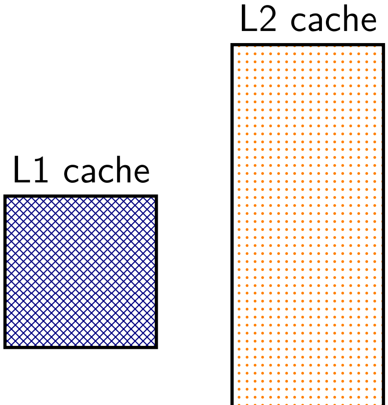
L2 inclusive of L1

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



L2 exclusive of L1

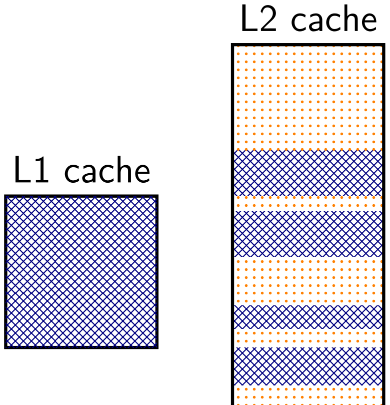
L2 contains different data than L1
adding to L1 must remove from L2
probably evicting from L1 adds to L2



inclusive versus exclusive

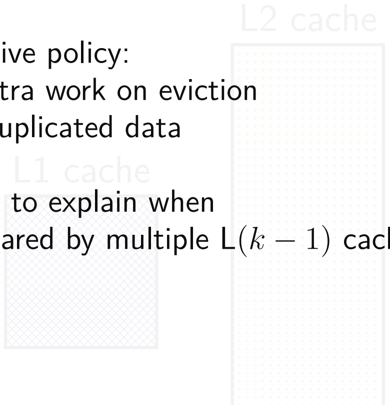
L2 inclusive of L1

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



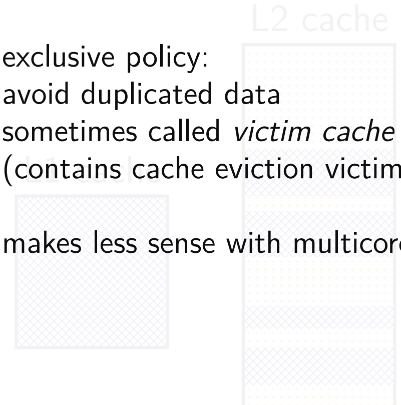
inclusive policy:
no extra work on eviction
but duplicated data

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

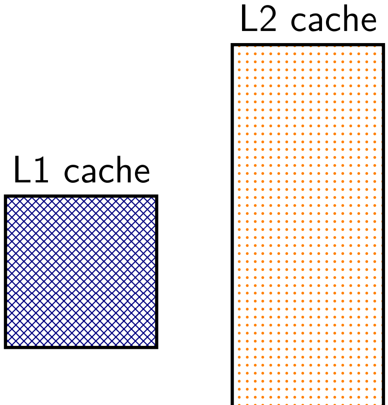


exclusive policy:
avoid duplicated data
sometimes called *victim cache*
(contains cache eviction victims)

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



Tag-Index-Offset formulas (direct-mapped)

(formulas derivable from prior slides)

$S = 2^s$	number of sets
s	(set) index bits
$B = 2^b$	block size
b	(block) offset bits
m	memory addreses bits
$t = m - (s + b)$	tag bits
$C = B \times S$	cache size (if direct-mapped)

Tag-Index-Offset formulas (direct-mapped)

(formulas derivable from prior slides)

$S = 2^s$	number of sets
s	(set) index bits
$B = 2^b$	block size
b	(block) offset bits
m	memory addreses bits
$t = m - (s + b)$	tag bits
$C = B \times S$	cache size (if direct-mapped)