

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 attacker'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/probing entries also traversed for other values

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

cache operation (read)

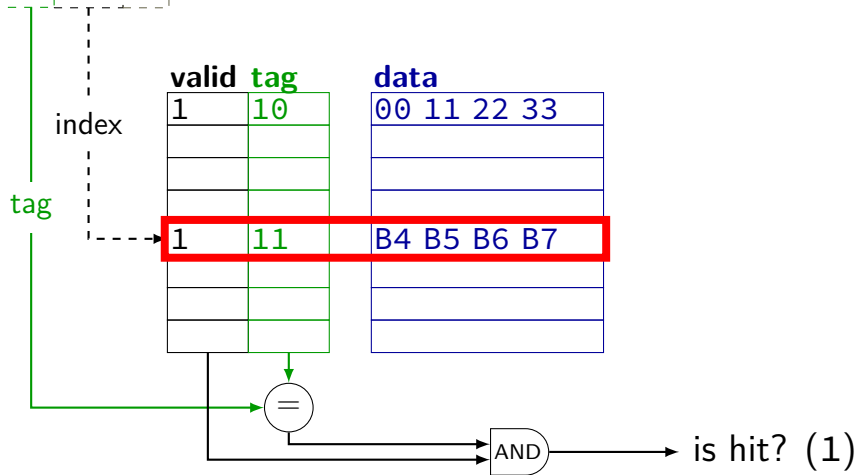
0b1110010

index

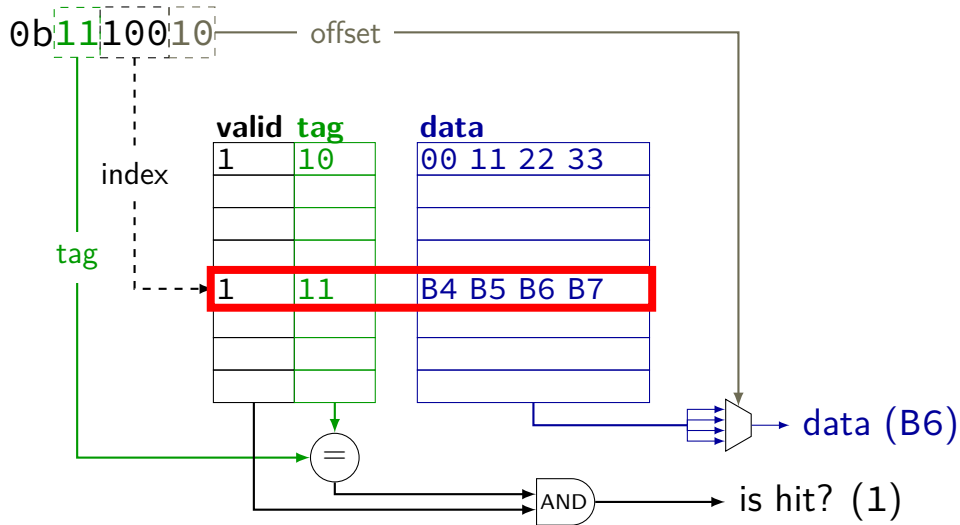
valid	tag	data
1	10	00 11 22 33
1	11	B4 B5 B6 B7

cache operation (read)

0b1110010



cache operation (read)



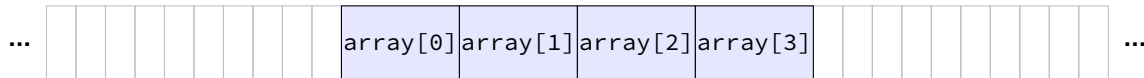
C and cache misses (warmup 1)

```
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 possibilities



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

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), and array[0] at beginning of cache block.

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

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?

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?

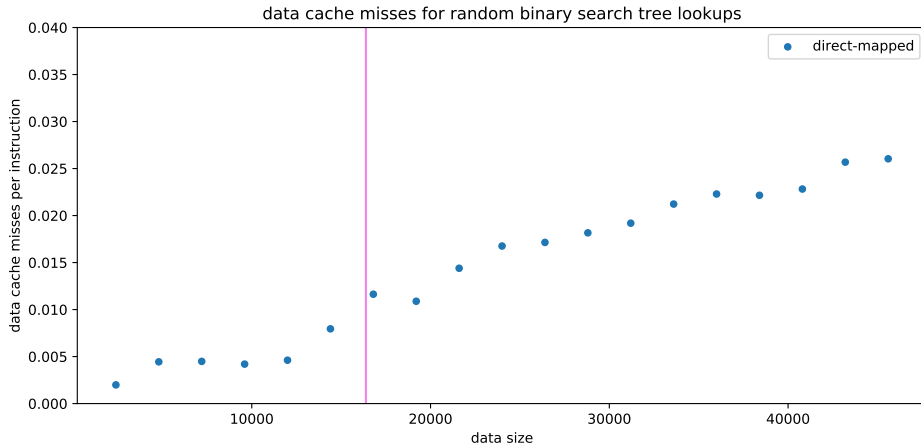
arrays and cache misses (3)

```
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).

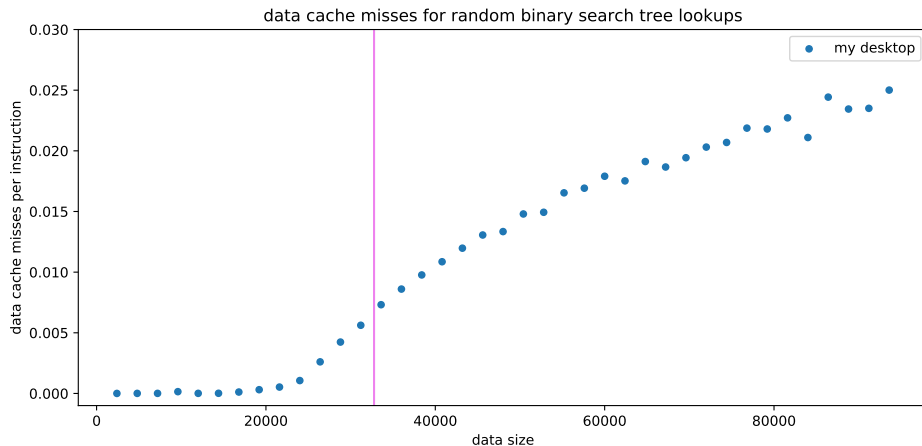
How many *data cache misses* on initially empty 2KB direct-mapped cache with 16B cache blocks?

simulated misses: BST lookups



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

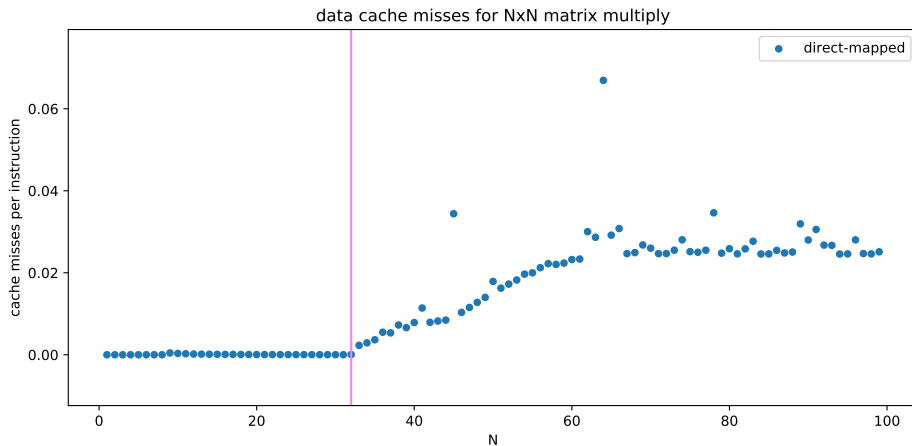
actual misses: BST lookups



(actual 32KB more complex data cache)

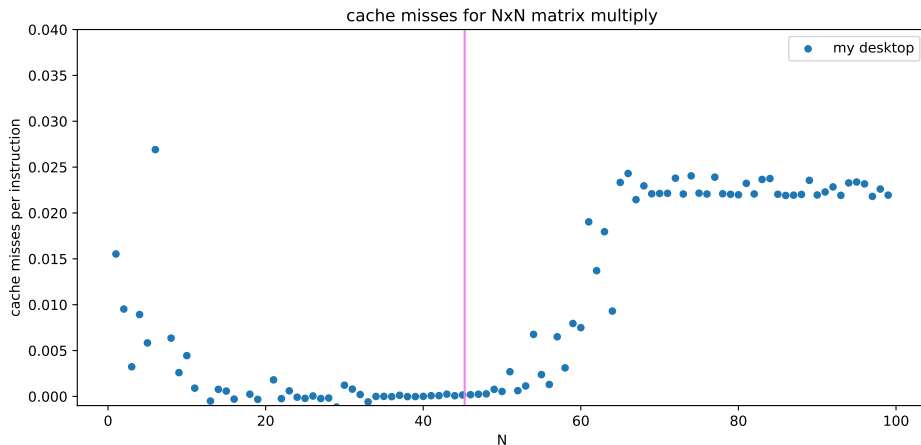
(only one set of measurements + other things on machine + excluding initial load)

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 $\# \text{ sets} \times \text{bytes/set}$

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 $\# \text{ sets} \times \text{bytes/set}$

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

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

way 0

way 1

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)	

tag indexoffset

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)	

tag index offset

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)	

tag indexoffset

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)	

tag index offset

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)	

tag index offset

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)	

tag indexoffset

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!

tag indexoffset

adding associativity

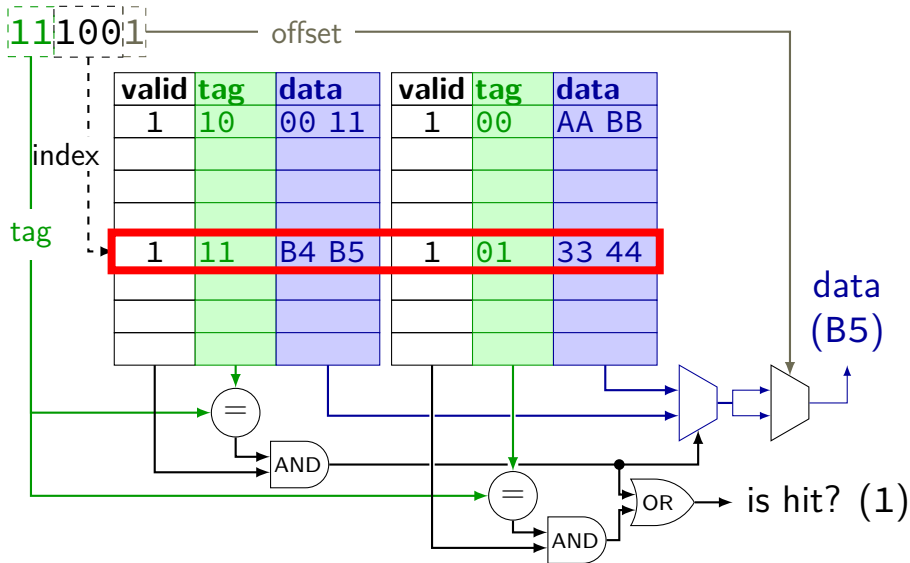
2-way set associative, 2 byte blocks, 2 sets

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		

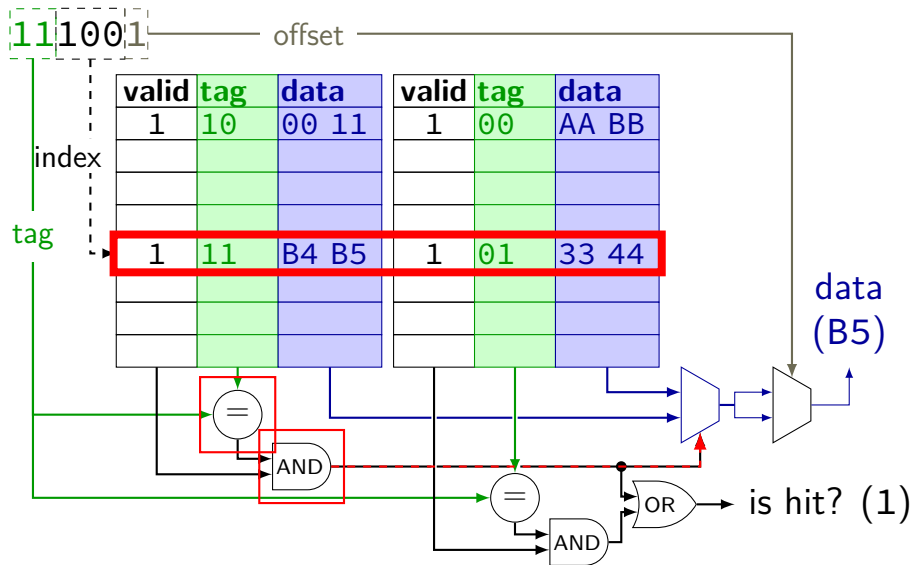
address (hex)	result
00000000 (00)	miss
00000001 (01)	hit
01100011 (63)	miss
01100001 (61)	miss
01100010 (62)	hit
00000000 (00)	hit
01100100 (64)	miss

tag index offset

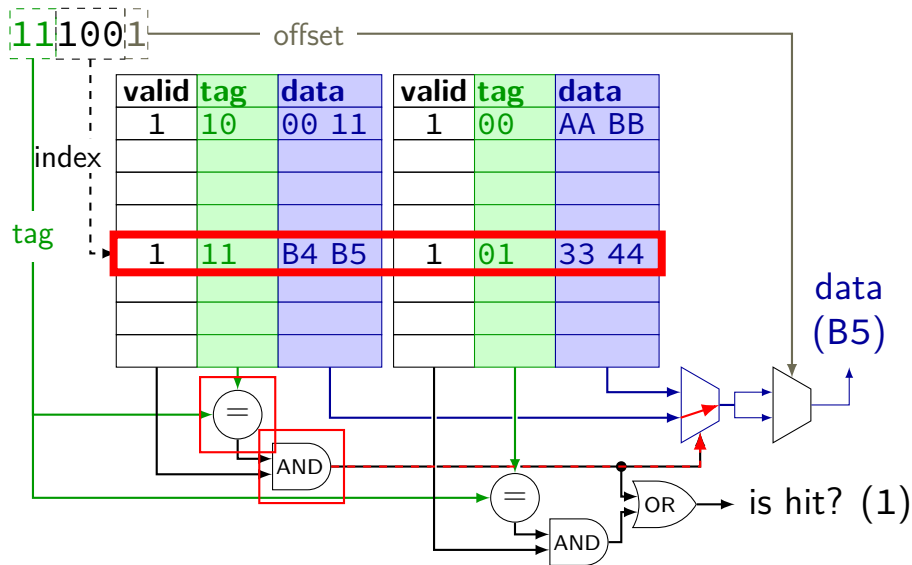
cache operation (associative)



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)	miss
00000001 (01)	hit
01100011 (63)	miss
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

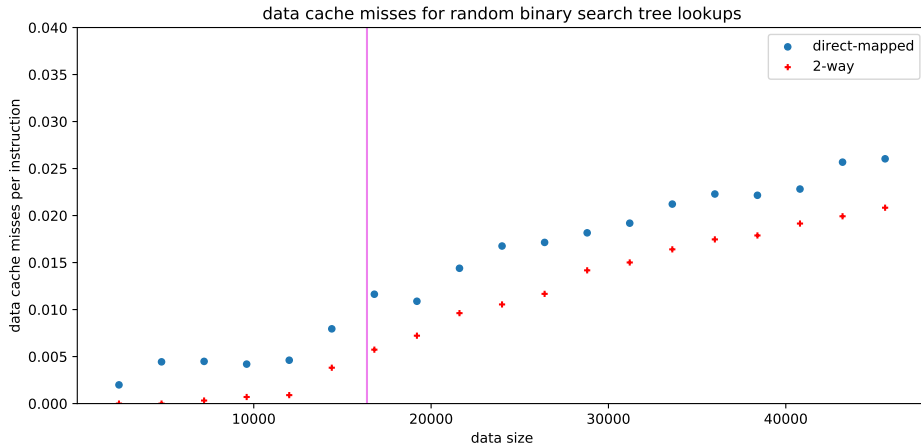
associativity terminology

direct-mapped — one block per set

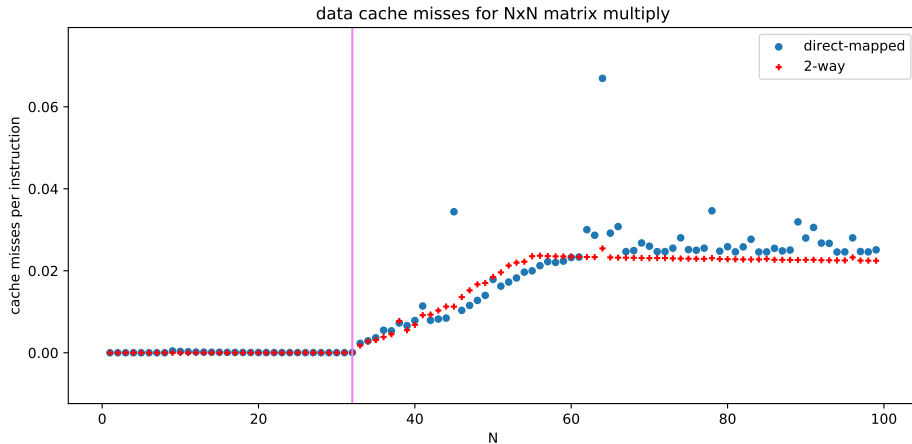
E -way set associative — E blocks per set
 E ways in the cache

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 addresses 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 addresses 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

T-I-O results

T-I-O: splitting

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];
```

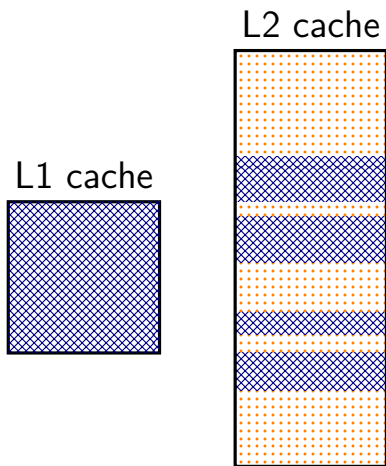
Assume everything but array is kept in registers (and the compiler does not do anything funny).

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

inclusive versus exclusive

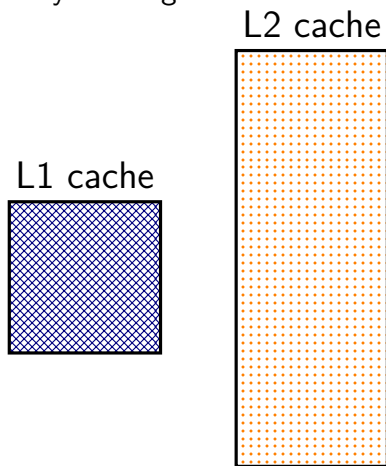
L2 inclusive of L1

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



L2 exclusive of L1

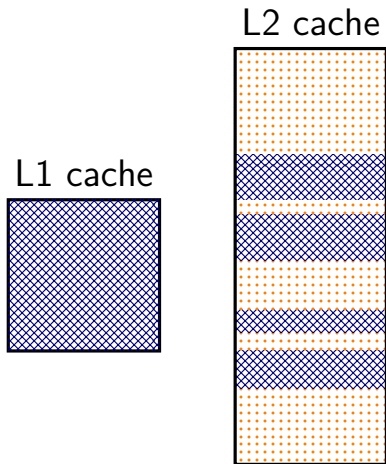
L2 contains different data than L1
adding to L1 must remove from L2
probably evicting from L1 adds to L2



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

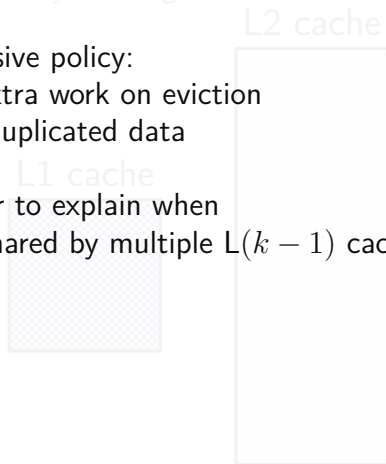
L2 contains different data than L1
adding to L1 must remove from L2
probably evicting from L1 adds to L2

inclusive policy:

no extra work on eviction
but duplicated data

easier to explain when

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



inclusive versus exclusive

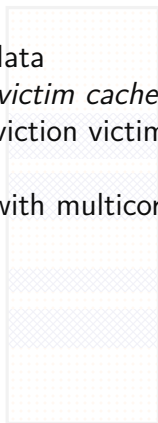
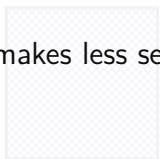
L2 inclusive of L1

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

L2 cache

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

makes less sense with multicore

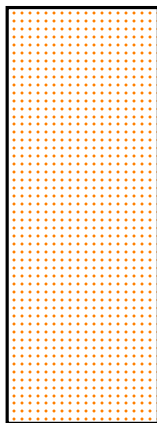
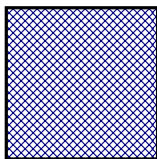


L2 exclusive of L1

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

L2 cache

L1 cache



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 addresses 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 addresses bits

$t = m - (s + b)$ tag bits

$C = B \times S$ **cache size** (if direct-mapped)