last time

operation of direct mapped caches

- computing beginning of blocks (offset bits = 0 always)
- always bring in value on read (temporal locality assumption)

calculating tag/index/offset splits

bad luck: two values accidentally map to same set?

E-way set associative cache

- E blocks per set (row)
- $E = 1$ = direct-mapped
- E = maximum = fully associative (1 set, 0 index bits)
- kinda like E direct-mapped caches together

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
  - lots of bookkeeping bits+time
- 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

exercise (1)

1 set, 5-way cache, 4 byte blocks

access pattern:

0x0, 0x1000, 0x4, 0x1004, 0x0, 0x1008, 0x4, 0x100C,
0x0, 0x1010, 0x4, 0x1014, 0x0, 0x1018, 0x4, 0x101C,
0x0, 0x1020, 0x4, 0x1024, 0x0, 0x1028, 0x4, 0x102C,
...

how does each do?

- LRU
- FIFO
- random

LRU

access	MRU ⟷ LRU
0x0	0x0-3
0x1000	0x1000-3, 0x0-3
0x4	0x4-7, 0x1000-3, 0x0-3
0x1004	0x1004-7, 0x4-7, 0x1000-3, 0x0-3
0x0	0x0-3, 0x1004-7, 0x4-7, 0x1000-3
0x1008	0x1008-B, 0x0-3, 0x1004-7, 0x4-7, 0x1000-3
0x4	0x4-7, 0x1008-B, 0x0-3, 0x1004-7, 0x1000-3
0x100C	0x100C-F, 0x4-7, 0x1008-B, 0x0-3, 0x1004-7
0x0	0x0-3, 0x100C-F, 0x4-7, 0x1008-B, 0x1004-7
0x1010	0x1010-3, 0x0-3, 0x100C-F, 0x4-7, 0x1008-B
…	…

0x0/0x4: 100% hit rate (after `warmup')

others: 0% hit rate

LRU

access	MRU ⟷ LRU
* 0x0	0x0-3
0x1000	0x1000-3, 0x0-3
* 0x4	0x4-7, 0x1000-3, 0x0-3
0x1004	0x1004-7, 0x4-7, 0x1000-3, 0x0-3
0x0	0x0-3, 0x1004-7, 0x4-7, 0x1000-3
0x1008	0x1008-B, 0x0-3, 0x1004-7, 0x4-7, 0x1000-3
0x4	0x4-7, 0x1008-B, 0x0-3, 0x1004-7, 0x1000-3
0x100C	0x100C-F, 0x4-7, 0x1008-B, 0x0-3, 0x1004-7
0x0	0x0-3, 0x100C-F, 0x4-7, 0x1008-B, 0x1004-7
0x1010	0x1010-3, 0x0-3, 0x100C-F, 0x4-7, 0x1008-B
…	…

0x0/0x4: 100% hit rate (after `warmup')

others: 0% hit rate

FIFO (1)

access	last in ⟷ first in
0x0	0x0-3
0x1000	0x1000-3, 0x0-3
0x4	0x4-7, 0x1000-3, 0x0-3
0x1004	0x1004-7, 0x4-7, 0x1000-3, 0x0-3
0x0	0x1004-7, 0x4-7, 0x1000-3, 0x0-3
0x1008	0x1008-B, 0x1004-7, 0x4-7, 0x1000-3, 0x0-3
0x4	0x1008-B, 0x1004-7, 0x4-7, 0x1000-3, 0x0-3
0x100C	0x100C-F, 0x1008-B, 0x1004-7, 0x4-7, 0x1000-3
0x0	0x0-3, 0x100C-F, 0x1008-B, 0x1004-7, 0x4-7
0x1010	0x1010-3, 0x0-3, 0x100C-F, 0x1008-B, 0x1004-7
0x4	0x4-7, 0x1010-3, 0x0-3, 0x100C-F, 0x1008-B
0x1014	0x1014-7, 0x4-7, 0x1010-3, 0x0-3, 0x100C-F
0x0	0x1014-7, 0x4-7, 0x1010-3, 0x0-3, 0x100C-F
0x1018	0x1018-B, 0x1014-7, 0x4-7, 0x1010-3, 0x0-3
0x4	0x1018-B, 0x1014-7, 0x4-7, 0x1010-3, 0x0-3
0x101C	0x101C-F, 0x1018-B, 0x1014-7, 0x4-7, 0x1010-3
0x0	0x0-3, 0x101C-F, 0x1018-B, 0x1014-7, 0x4-7

FIFO (1)

```
access  last in ⟷ first in
M 0x0   0x0-3
0x1000  0x1000-3, 0x0-3
M 0x4   0x4-7, 0x1000-3, 0x0-3
0x1004  0x1004-7, 0x4-7, 0x1000-3, 0x0-3
H 0x0   0x1004-7, 0x4-7, 0x1000-3, 0x0-3
0x1008  0x1008-B, 0x1004-7, 0x4-7, 0x1000-3, 0x0-3
H 0x4   0x1008-B, 0x1004-7, 0x4-7, 0x1000-3, 0x0-3
0x100C  0x100C-F, 0x1008-B, 0x1004-7, 0x4-7, 0x1000-3
M 0x0   0x0-3, 0x100C-F, 0x1008-B, 0x1004-7, 0x4-7
0x1010  0x1010-3, 0x0-3, 0x100C-F, 0x1008-B, 0x1004-7
M 0x4   0x4-7, 0x1010-3, 0x0-3, 0x100C-F, 0x1008-B
0x1014  0x1014-7, 0x4-7, 0x1010-3, 0x0-3, 0x100C-F
H 0x0   0x1014-7, 0x4-7, 0x1010-3, 0x0-3, 0x100C-F
0x1018  0x1018-B, 0x1014-7, 0x4-7, 0x1010-3, 0x0-3
H 0x4   0x1018-B, 0x1014-7, 0x4-7, 0x1010-3, 0x0-3
0x101C  0x101C-F, 0x1018-B, 0x1014-7, 0x4-7, 0x1010-3
M 0x0   0x0-3, 0x101C-F, 0x1018-B, 0x1014-7, 0x4-7
```

FIFO (2)

access	last in ⟷ first in
…	…
0x100C	0x100C-F, 0x1008-B, 0x1004-7, 0x4-7, 0x1000-3
0x0	0x0-3, 0x100C-F, 0x1008-B, 0x1004-7, 0x4-7
0x1010	0x1010-3, 0x0-3, 0x100C-F, 0x1008-B, 0x1004-7
0x4	0x4-7, 0x1010-3, 0x0-3, 0x100C-F, 0x1008-B
0x1014	0x1014-7, 0x4-7, 0x1010-3, 0x0-3, 0x100C-F
0x0	0x1014-7, 0x4-7, 0x1010-3, 0x0-3, 0x100C-F
0x1018	0x1018-B, 0x1014-7, 0x4-7, 0x1010-3, 0x0-3
0x4	0x1018-B, 0x1014-7, 0x4-7, 0x1010-3, 0x0-3
0x101C	0x101C-F, 0x1018-B, 0x1014-7, 0x4-7, 0x1010-3
0x0	0x0-3, 0x101C-F, 0x1018-B, 0x1014-7, 0x4-7
0x1020	0x1020-3, 0x0-3, 0x101C-F, 0x1018-B, 0x1014-7
0x4	0x0-4, 0x1020-3, 0x101C-F, 0x1018-B, 0x1014-7
…	…

0x0/0x4: approx 50% hit rate (after `warmup')
others: 0% hit rate

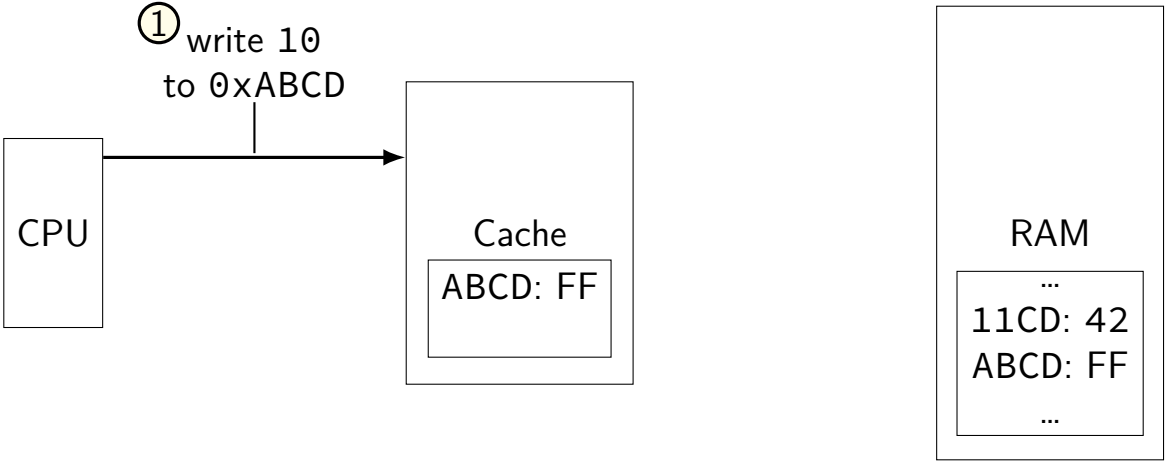
FIFO (2)

	access	last in ⟷ first in
…		…
	0x100C	0x100C-F, 0x1008-B, 0x1004-7, 0x4-7, 0x1000-3
M	0x0	0x0-3, 0x100C-F, 0x1008-B, 0x1004-7, 0x4-7
	0x1010	0x1010-3, 0x0-3, 0x100C-F, 0x1008-B, 0x1004-7
M	0x4	0x4-7, 0x1010-3, 0x0-3, 0x100C-F, 0x1008-B
	0x1014	0x1014-7, 0x4-7, 0x1010-3, 0x0-3, 0x100C-F
H	0x0	0x1014-7, 0x4-7, 0x1010-3, 0x0-3, 0x100C-F
	0x1018	0x1018-B, 0x1014-7, 0x4-7, 0x1010-3, 0x0-3
H	0x4	0x1018-B, 0x1014-7, 0x4-7, 0x1010-3, 0x0-3
	0x101C	0x101C-F, 0x1018-B, 0x1014-7, 0x4-7, 0x1010-3
M	0x0	0x0-3, 0x101C-F, 0x1018-B, 0x1014-7, 0x4-7
	0x1020	0x1020-3, 0x0-3, 0x101C-F, 0x1018-B, 0x1014-7
M	0x4	0x0-4, 0x1020-3, 0x101C-F, 0x1018-B, 0x1014-7
…		…

0x0/0x4: approx 50% hit rate (after `warmup')
others: 0% hit rate

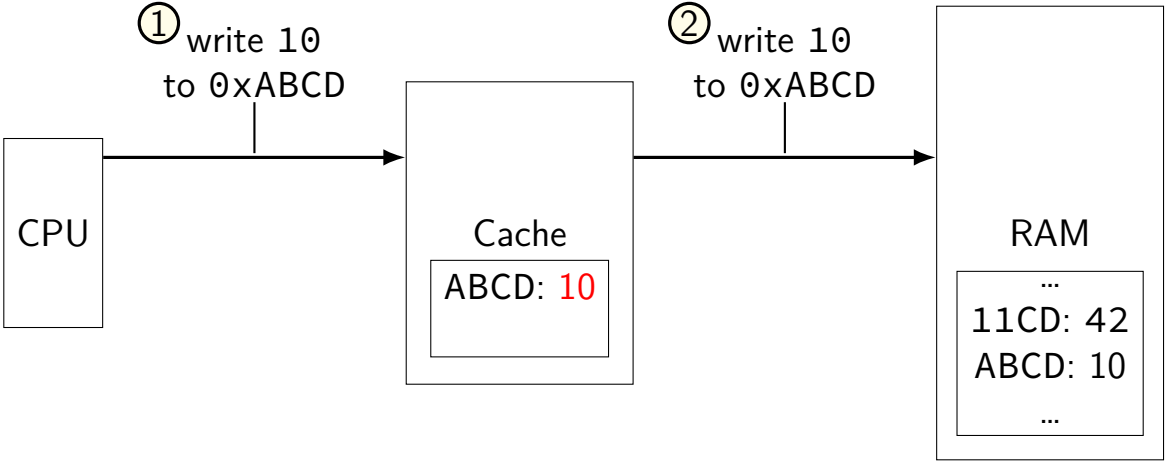
write-through v. write-back

option 1: write-through



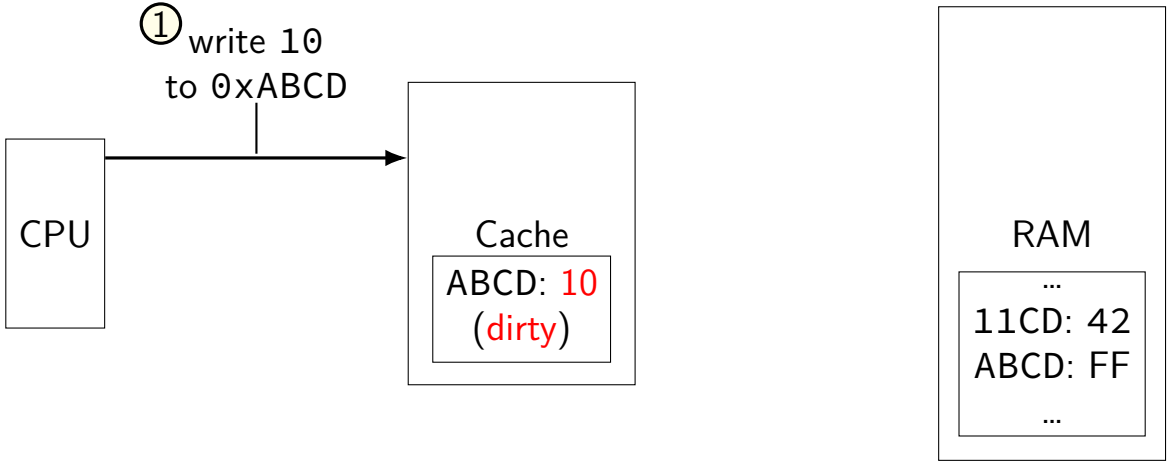
write-through v. write-back

option 1: write-through



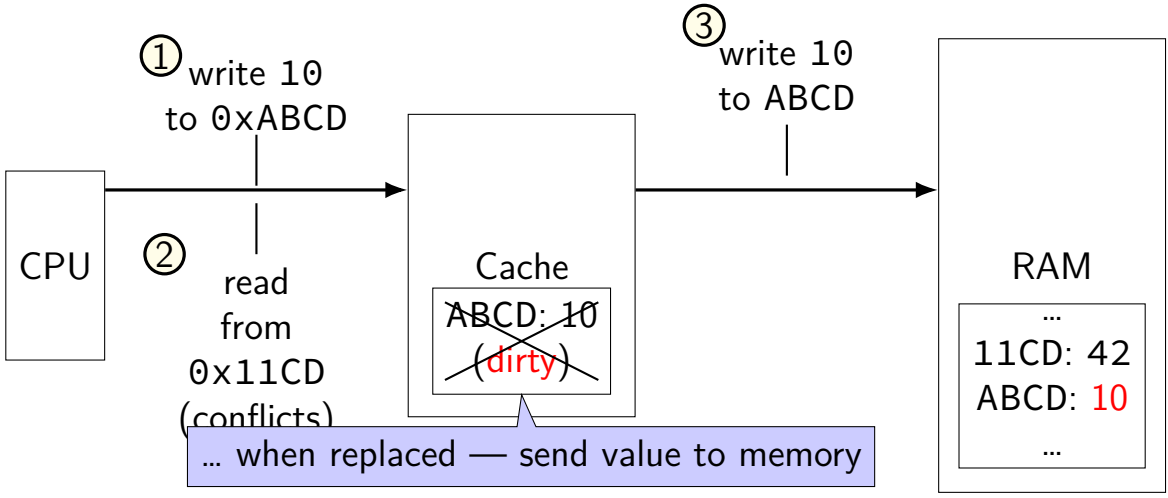
write-through v. write-back

option 2: write-back

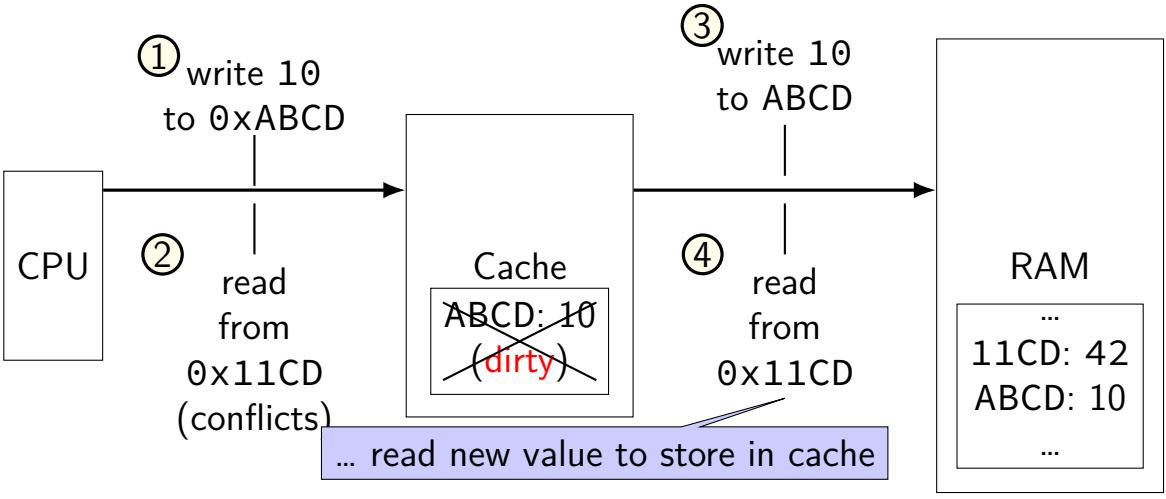


write-through v. write-back

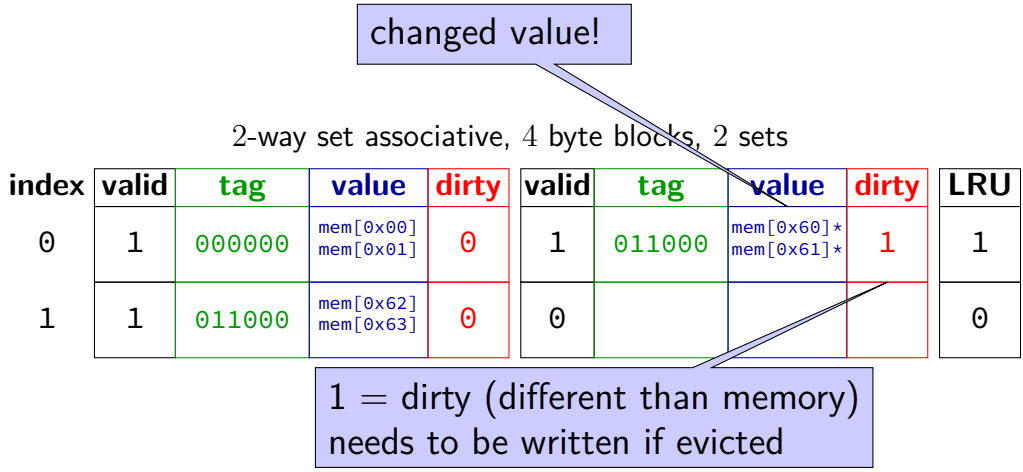
option 2: write-back



write-through v. write-back



writeback policy



allocate on write?

processor writes less than whole cache block

block not yet in cache

two options:

write-allocate

fetch rest of cache block, replace written part
(then follow write-through or write-back policy)

write-no-allocate

don't use cache at all (send write to memory *instead*)
guess: not read soon?

write-allocate

2-way set associative, LRU, writeback

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

writing $\widehat{0}$xFF into address 0x04?

index 0, tag 000001

write-allocate

2-way set associative, LRU, writeback

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

writing $\widehat{0}$xFF into address 0x04?

index 0, tag 000001

step 1: find least recently used block

write-allocate

2-way set associative, LRU, writeback

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

writing $\widehat{0}$xFF into address 0x04?

index 0, tag 000001

step 1: find least recently used block

step 2: possibly writeback old block

write-allocate

2-way set associative, LRU, writeback

index	valid	tag	value	dirty	valid	tag	value	dirty	LRU
0	1	000000	mem[0x00] mem[0x01]	0	1	000001	0xFF mem[0x05]	1	0
1	1	011000	mem[0x62] mem[0x63]	0	0				0

writing $\widehat{0}$xFF into address 0x04?

index 0, tag 000001

step 1: find least recently used block

step 2: possibly writeback old block

step 3a: read in new block – to get mem[0x05]

step 3b: update LRU information

write-no-allocate

2-way set associative, LRU, writeback

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

writing $\widehat{0}$xFF into address 0x04?

step 1: is it in cache yet?

step 2: no, just send it to memory

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]*	~~10~~	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

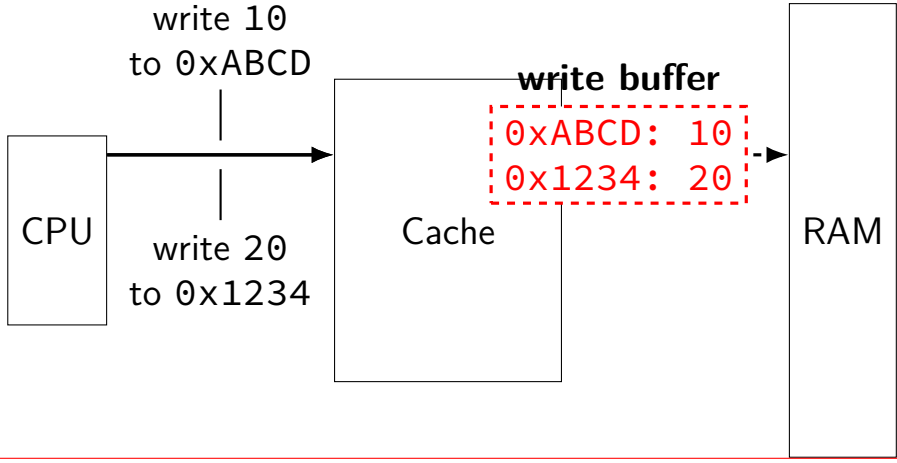
index	valid	tag	value	valid	tag	value	LRU
0	1	001100	mem[0x30] mem[0x31]	1	010000	mem[0x40] mem[0x41]	0
1	1	011000	mem[0x62] mem[0x63]	1	001100	mem[0x32] mem[0x33]	1

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

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 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

backup slides

exercise (2)

2 set, 3-way cache, 4 byte blocks

access pattern:

0x0, 0x1000, 0x4, 0x1004, 0x0, 0x1008, 0x4, 0x100C,
0x0, 0x1010, 0x4, 0x1014, 0x0, 0x1018, 0x4, 0x101C,
0x0, 0x1020, 0x4, 0x1024, 0x0, 0x1028, 0x4, 0x102C,
...

how does each do?

LRU
FIFO
random

exercise (2)

2 set, 3-way cache, 4 byte blocks

access pattern:

0x0, 0x1000, 0x4, 0x1004, 0x0, 0x1008, 0x4, 0x100C,
0x0, 0x1010, 0x4, 0x1014, 0x0, 0x1018, 0x4, 0x101C,
0x0, 0x1020, 0x4, 0x1024, 0x0, 0x1028, 0x4, 0x102C,
...

how does each do?

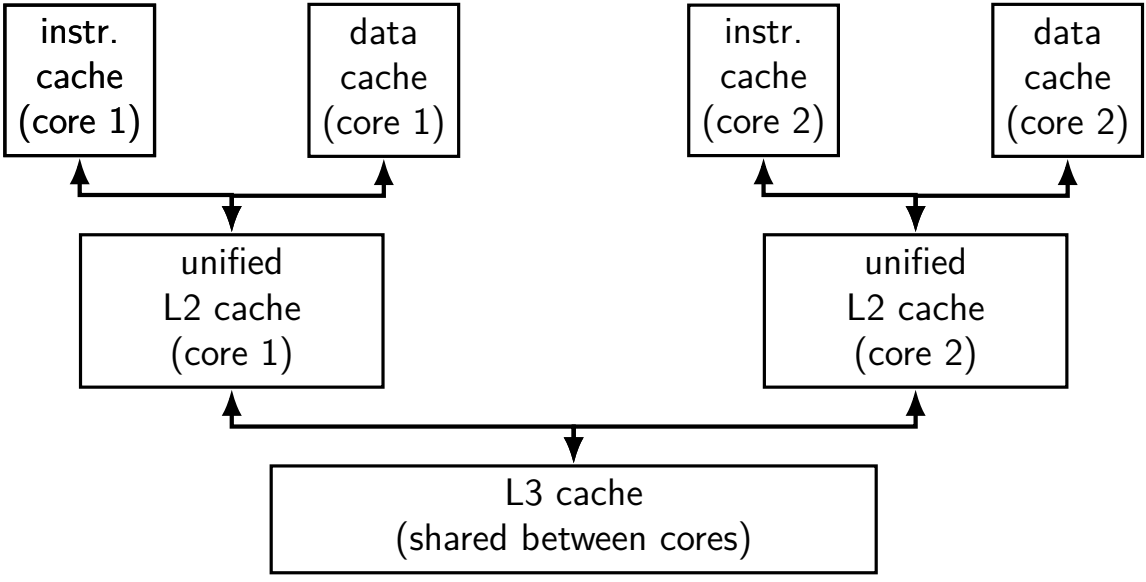
- LRU
- FIFO
- random

FIFO

access	last in ⟷ first in
0x0	0x0
0x1000	0x1000, 0x0
0x0	0x1000, 0x0
0x1008	0x1008, 0x1000, 0x0
0x0	0x1008, 0x1000, 0x0
0x1010	0x1010, 0x1008, 0x1000
0x0	0x0, 0x1010, 0x1008
0x1018	0x1018, 0x0, 0x1010
0x0	0x1018, 0x0, 0x1010
0x1020	0x1020, 0x1018, 0x0
0x0	0x1020, 0x1018, 0x0
0x1028	0x1028, 0x1020, 0x1018
0x0	0x0, 0x1028, 0x1020
…	…

0x0: 67% hit rate; others: 0% hit rate

split caches; multiple cores



hierarchy and instruction/data caches

typically separate data and instruction caches for L1

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

avoids instructions evicting data and vice-versa

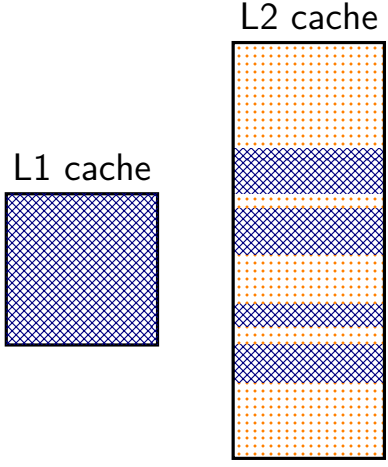
can optimize instruction cache for different access pattern

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

inclusive versus exclusive

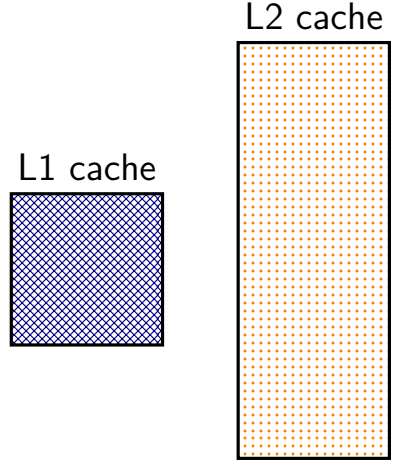
L2 inclusive of L1

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



L2 exclusive of L1

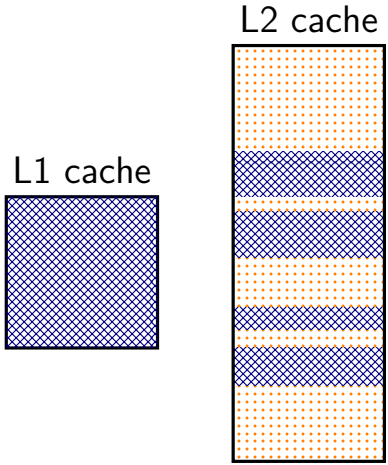
L2 contains different data than L1
adding to L1 must remove from L2
probably evicting from L1 adds to L2



inclusive versus exclusive

L2 inclusive of L1

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



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

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