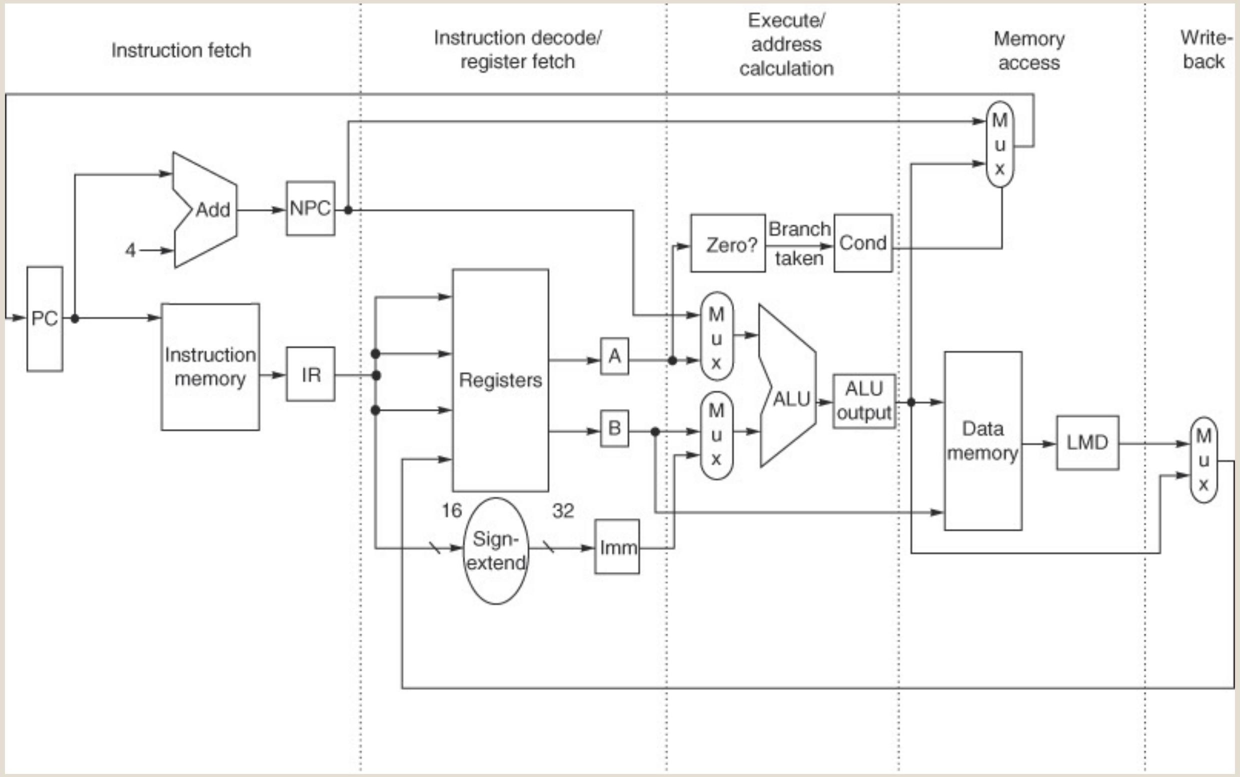
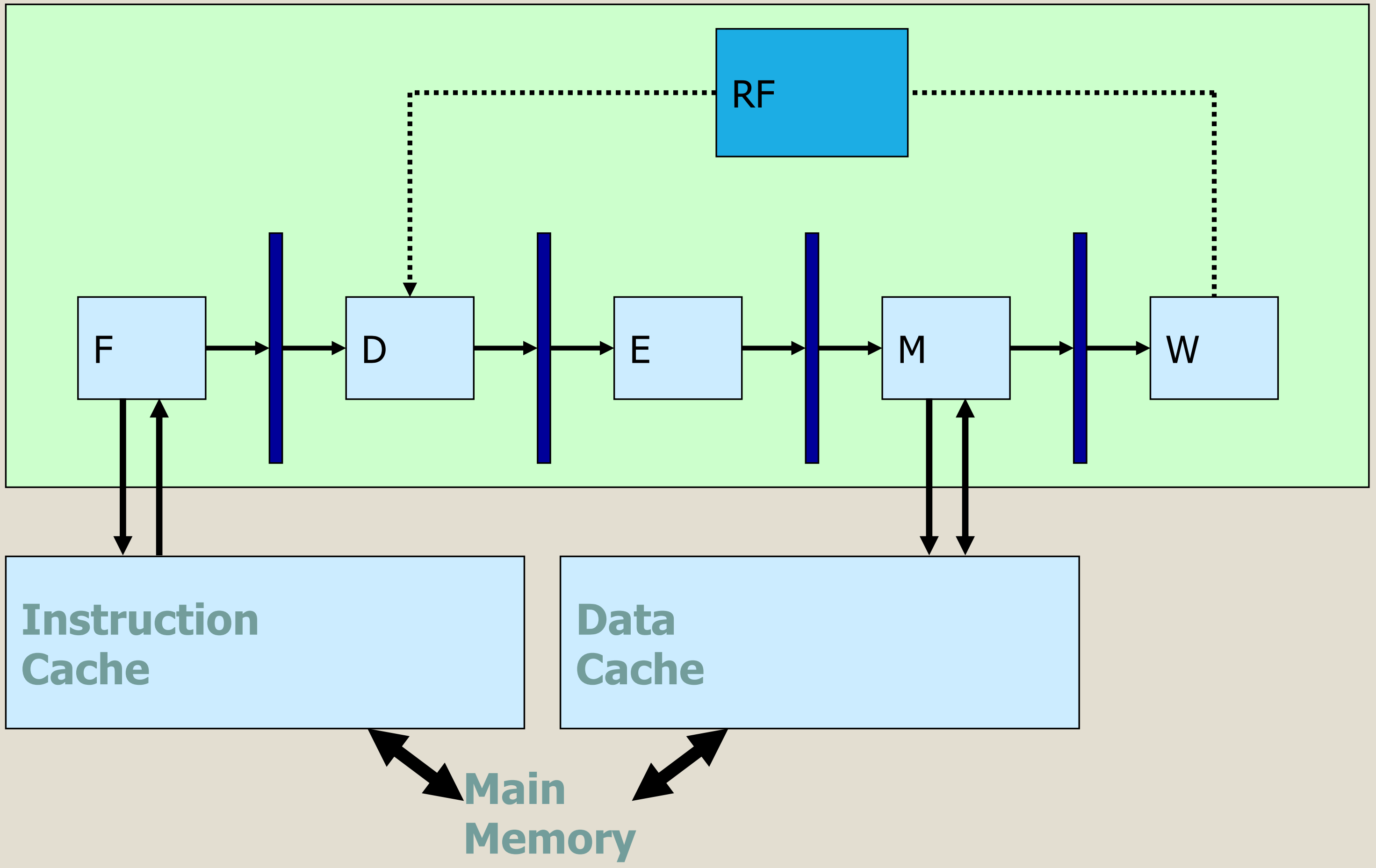
Caches


Instruction fetch
Instruction decode/ register fetch
Execute/ address calculation
Memory access
Write- back
PC
Add
4
NPC
Instruction memory
IR
Registers
A
B
16
Sign- extend
32
Imm
Zero?
Branch taken
Cond
M u x
ALU
ALU output
Data memory
LMD


RF
F
D
E
M
W
Instruction Cache
Data Cache
Main Memory

Why

1. Exploit spatial and temporal locality

2. Exploit memory technology

1: Locality

1. Spatial: different data, close in memory, are likely to be accessed together

2. Temporal: the same data is likely to be accessed multiple times in the near future

```
for(i=0; i<N; i++)
  a[i] = a[i] + c[j];
```

spatial: a[i], a[i+1], a[i+2] etc are accessed

temporal: c[j] is needed not just once, but N times

Note that exactly the same is true for instructions (not just for data)
In fact, we have a data cache, and an instruction cache

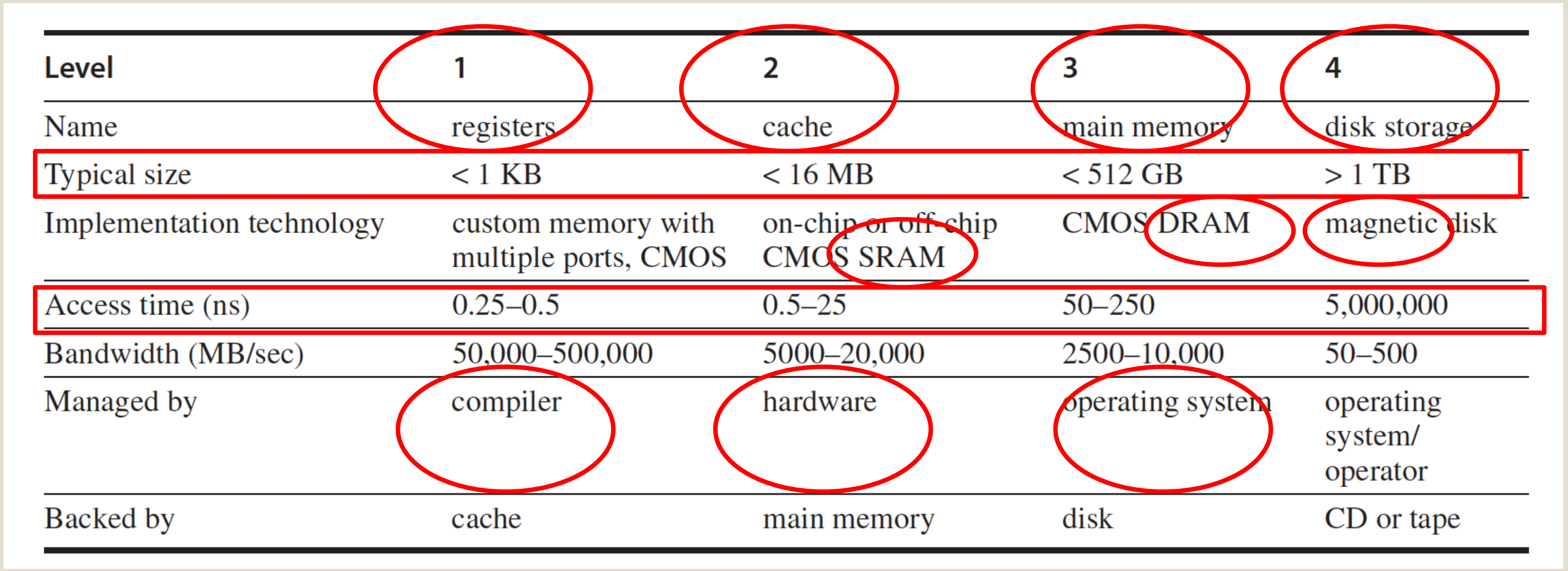
2: Memory Technology

1. Exploit memory technology

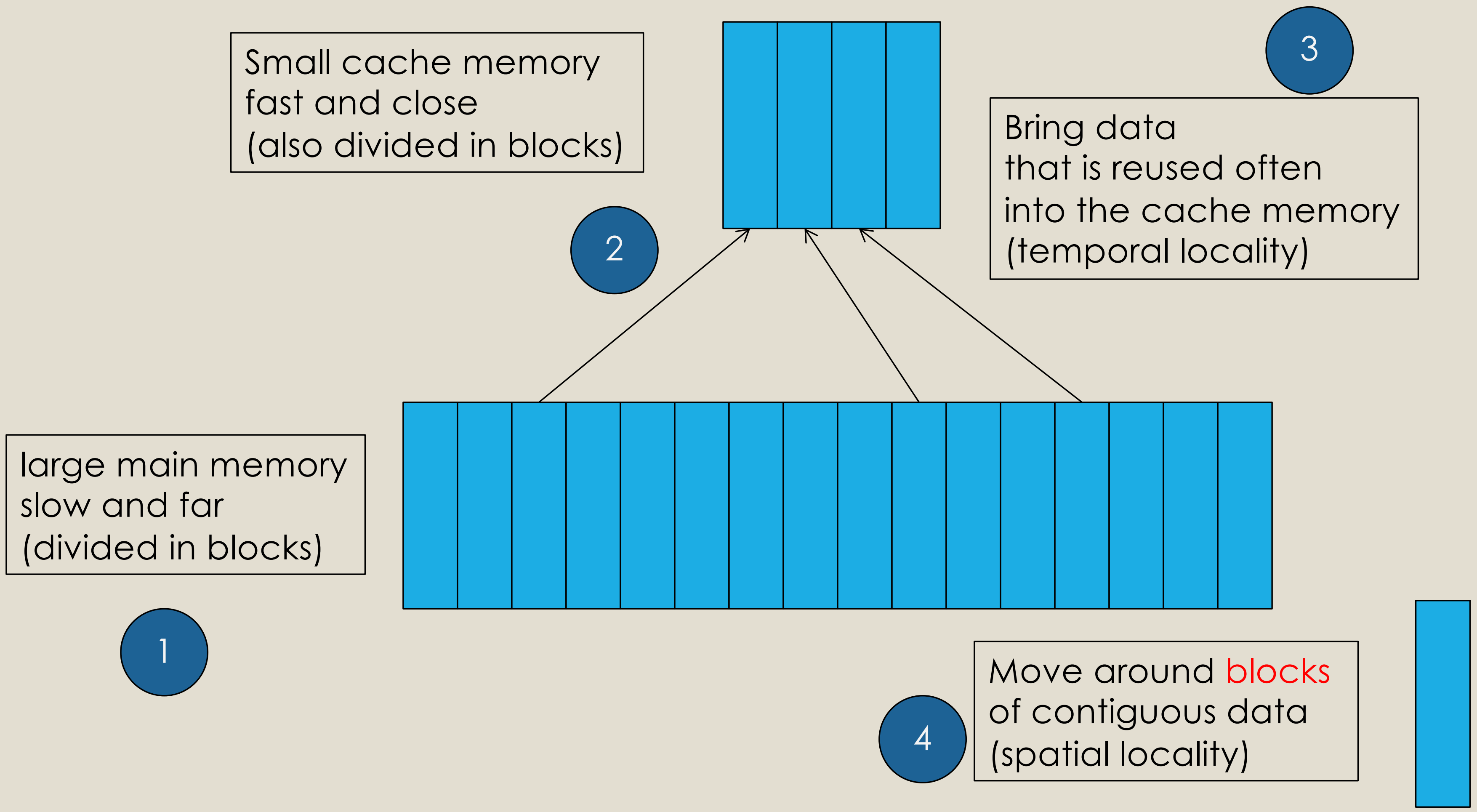
   Expensive memories have faster access

   → Build small and expensive memories to handle the frequent cases, use large and inexpensive ones for the less frequent cases

2: Memory Technology



Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	< 16 MB	< 512 GB	> 1 TB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25–0.5	0.5–25	50–250	5,000,000
Bandwidth (MB/sec)	50,000–500,000	5000–20,000	2500–10,000	50–500
Managed by	compiler	hardware	operating system	operating system/ operator
Backed by	cache	main memory	disk	CD or tape


1
large main memory
slow and far
(divided in blocks)
2
Small cache memory
fast and close
(also divided in blocks)
3
Bring data
that is reused often
into the cache memory
(temporal locality)
4
Move around blocks
of contiguous data
(spatial locality)

Four questions

1. Where can a block be placed (block placement)

2. How is a block found (block identification)

3. Which block should be replaced on a miss (block replacement)

4. What happens on a write (write strategy)

1: Block placement

- Direct mapped: each block has only one place it can appear in the cache

  The mapping is: (Block address) MOD (Number of blocks in the cache)

- Set associative: each block can be placed in a restrictive set of places in the cache

  If there are m elements in each set, it's called m-way set associative

  The mapping is: (Block address) MOD (Number of sets in the cache)

- Fully associative: each block can be put anywhere in the cache

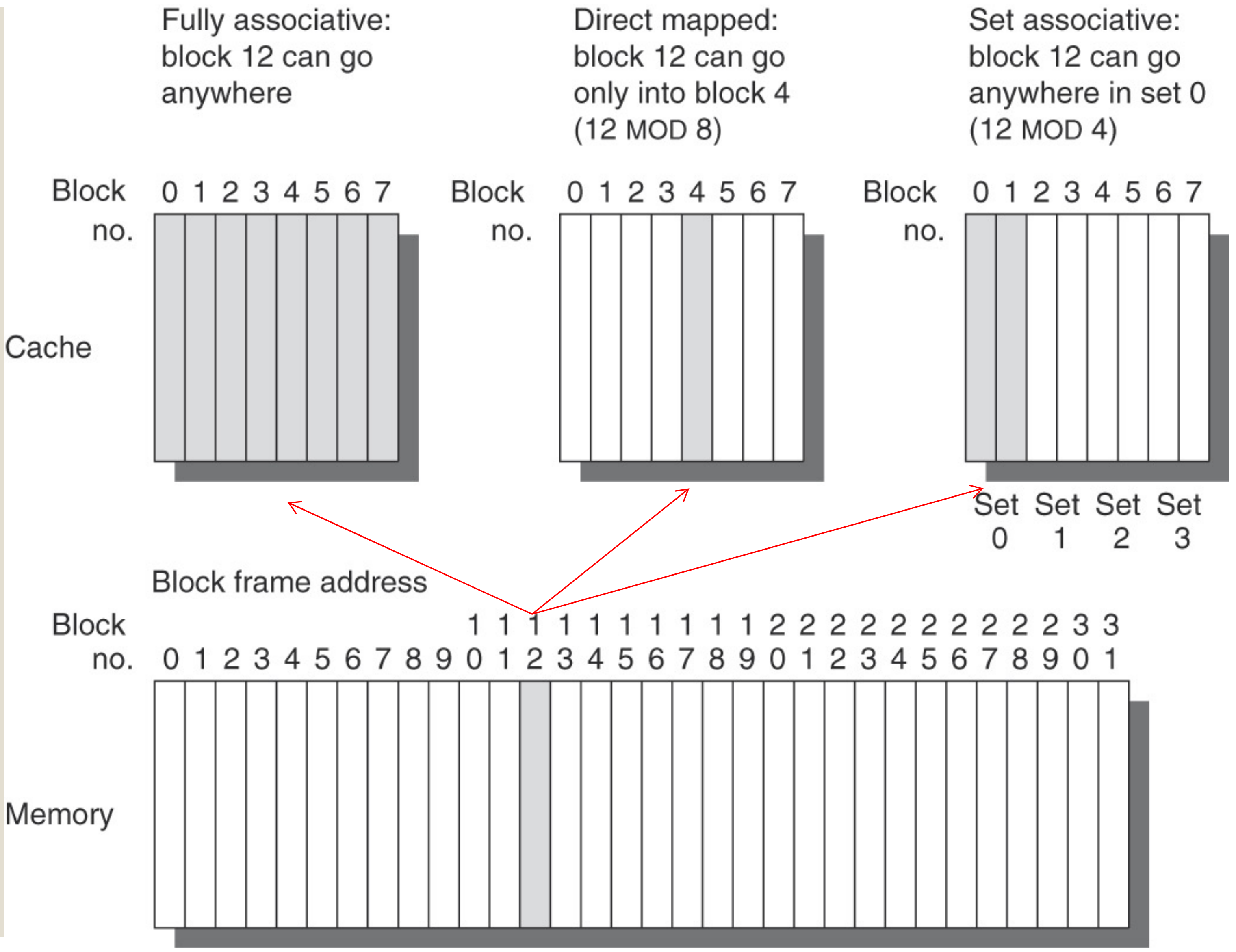
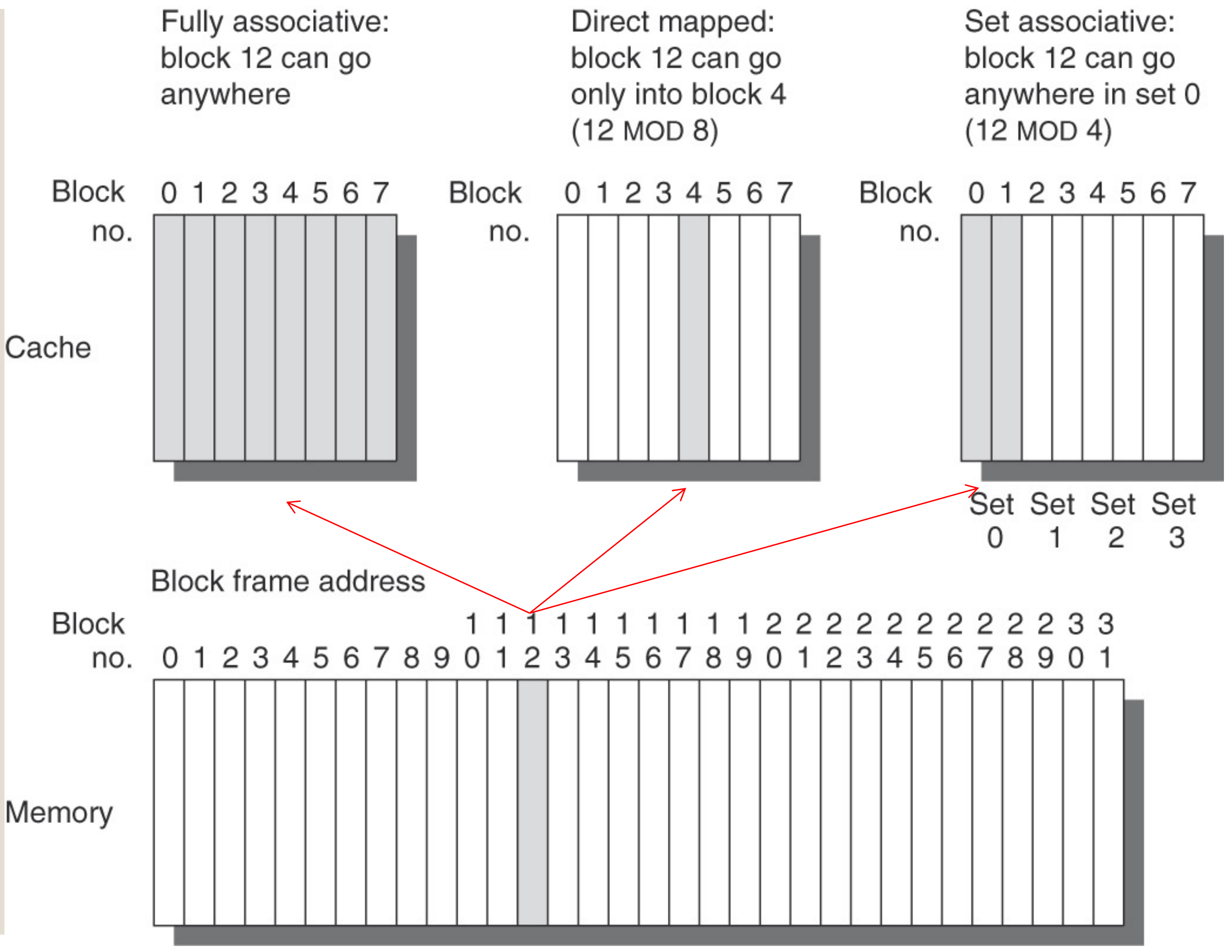

Fully associative: block 12 can go anywhere
Direct mapped: block 12 can go only into block 4 (12 MOD 8)
Set associative: block 12 can go anywhere in set 0 (12 MOD 4)
Block no.
0 1 2 3 4 5 6 7
Cache
Set 0
Set 1
Set 2
Set 3
Block frame address
Block no.
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Memory

Figure B.2 This example cache has eight block frames and memory has 32 blocks. The three options for caches are shown left to right. In fully associative, block 12 from the lower level can go into any of the eight block frames of the cache. With direct mapped, block 12 can only be placed into block frame 4 (12 modulo 8). Set associative, which has some of both features, allows the block to be placed anywhere in set 0 (12 modulo 4). With two blocks per set, this means block 12 can be placed either in block 0 or in block 1 of the cache. Real caches contain thousands of block frames, and real memories contain millions of blocks. The set associative organization in this Figure has four sets with two blocks per set, called *two-way set associative*.



Fully associative: block 12 can go anywhere
Direct mapped: block 12 can go only into block 4 (12 MOD 8)
Set associative: block 12 can go anywhere in set 0 (12 MOD 4)
Block no. 0 1 2 3 4 5 6 7
Cache
Set 0 Set 1 Set 2 Set 3
Block frame address
Block no. 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Memory

M-way set associative

- Direct mapped = 1-way set associative

- Set associative = m-way set associative (where m is the number of blocks per set, in the cache)

- Fully associative = f-way set associative (where f is the number of blocks, in the cache)

In practice:

- First-level caches tend to have low-associativity:
  - 1, 2, or 4-way associative

      (with some exceptions, e.g. intel i7-i9: 1L Dcache is 8-way)

- why? Because high-level associativity comes at a cost

see next point

Caches: four questions

1. Where can a block be placed (block placement)

2. How is a block found (block identification)

3. Which block should be replaced on a miss (block replacement)

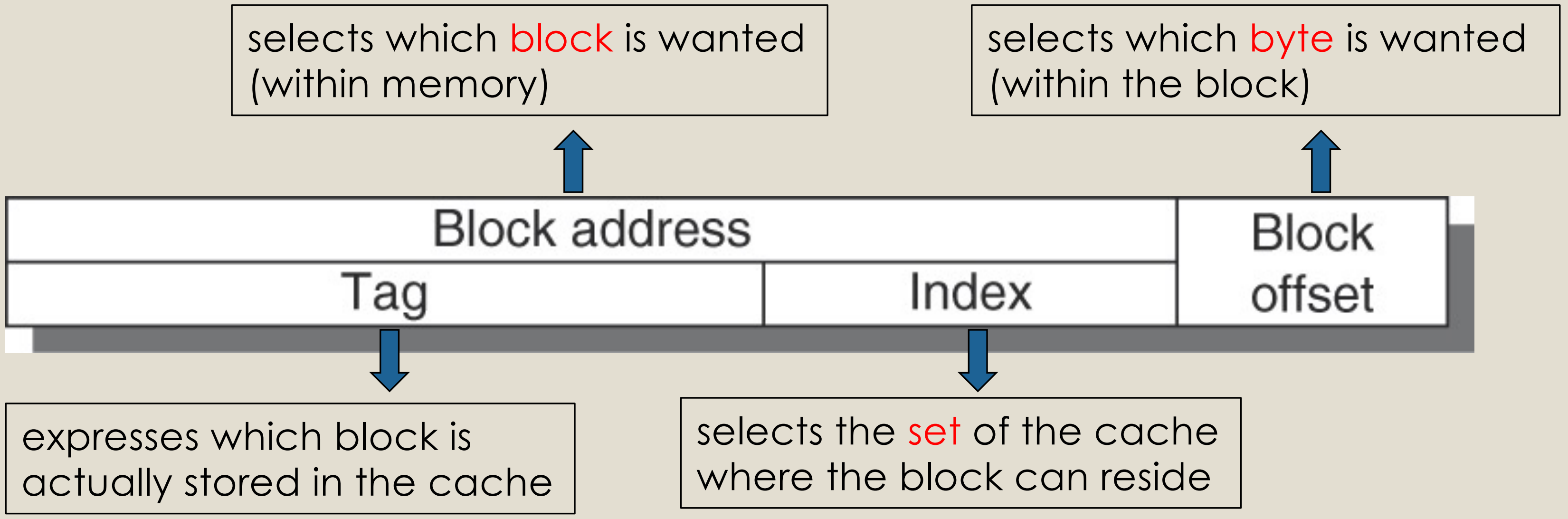
4. What happens on a write (write strategy)

2: Block identification

- Question number 2 was: how is a block found in the cache?

- By checking the address tag of each block in the right set
  - Let’s see what this means

The structure of a memory address


selects which block is wanted (within memory)
selects which byte is wanted (within the block)
Block address
Tag
Index
Block offset
expresses which block is actually stored in the cache
selects the set of the cache where the block can reside

The first division is between the block address and offset

The block offset is the address of the desired byte within the block

The block address can be further divided into tag and index

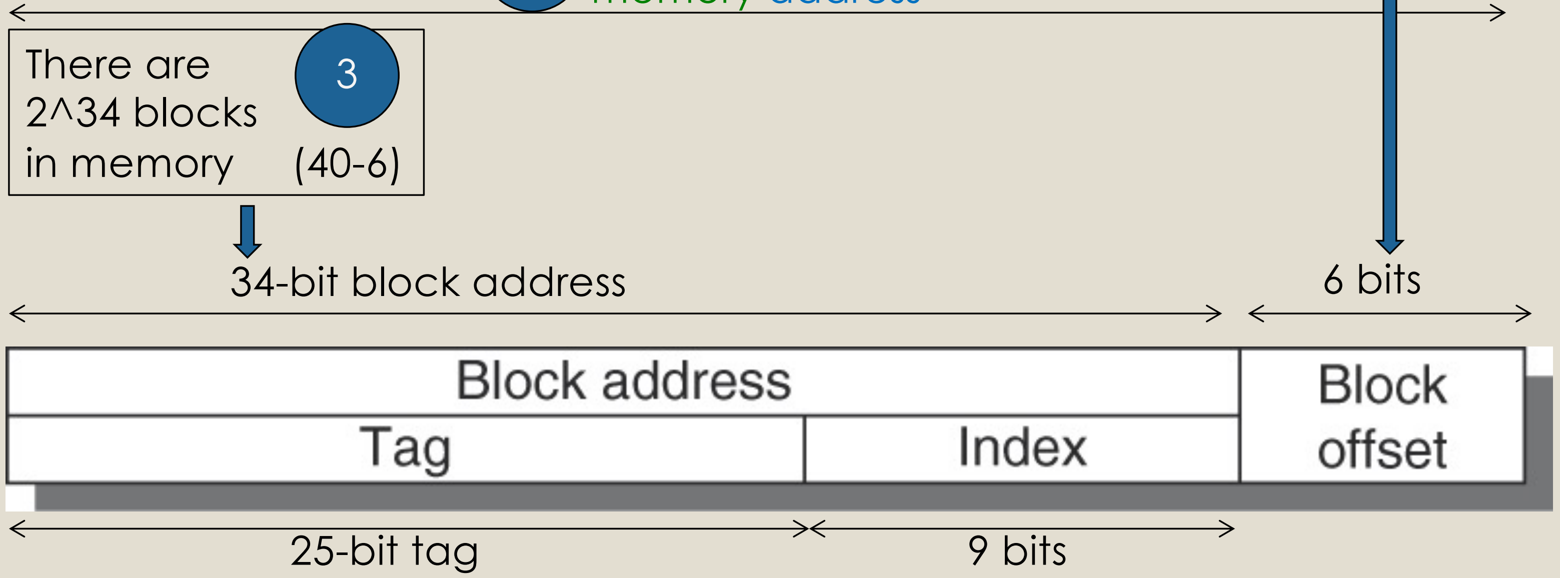
Example: Opteron

each block has 64 bytes 2

64 = 2^6

1 40-bit physical memory address

There are 2^34 blocks in memory 3 (40-6)



The cache contains 64 K bytes of data in 64-byte blocks 4

How many blocks? → 1024 blocks .

The cache is two-way set associative

1024 blocks / 2
(for 2-way set associative) =
512 sets
→ 9 bits of index
to select among the sets 5

Back to block identification

- How do I find out whether a certain block is in the cache or not?

$\rightarrow$

the tag of every cache block

that might contain the desired information

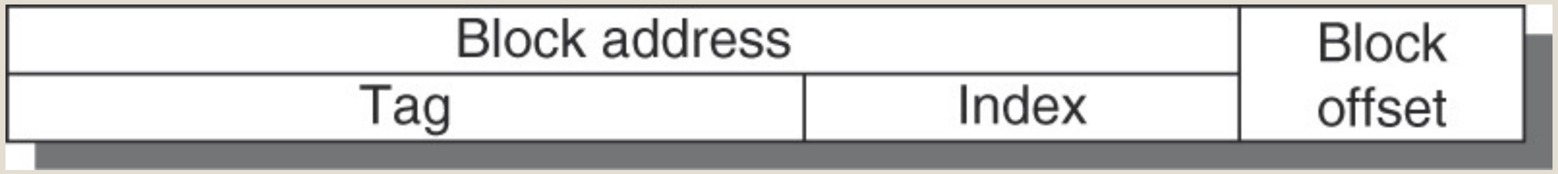
is checked, to see whether it matches

the tag of the memory address

i.e.: in the right set

Why just the tag?

- No point in checking the offset: the whole block is there (or not there) anyway
- No point in checking the index: we are already checking the tag of the blocks within the set selected by the index



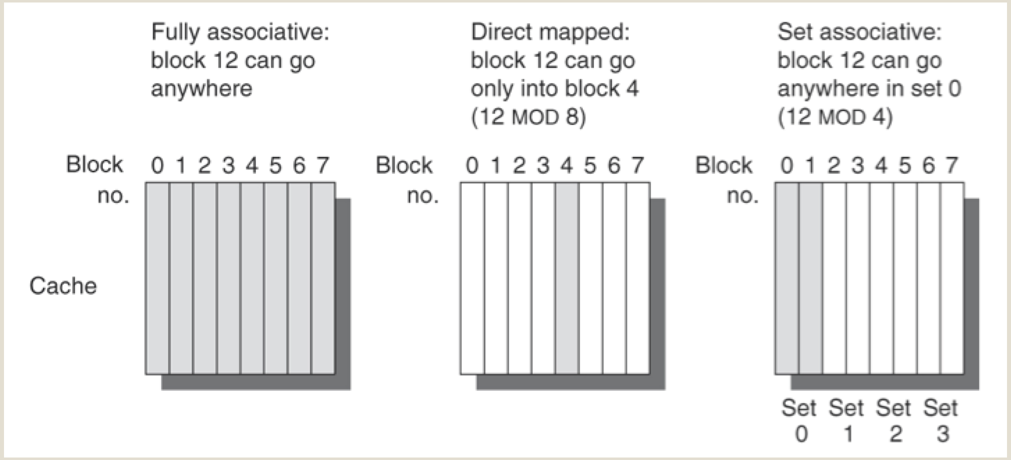
Performance

- The tags of all blocks in the same set must be checked
- The larger a set (=higher associativity =less bits in the index) → the more blocks to check

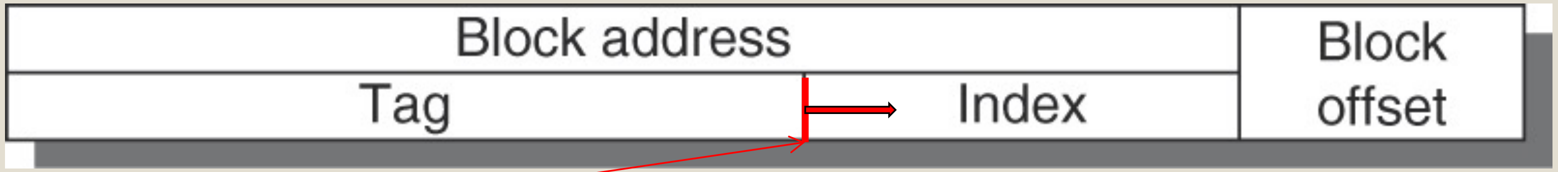
→ this limits the amount

of associativity

one should have



Fully associative: block 12 can go anywhere
Direct mapped: block 12 can go only into block 4 (12 MOD 8)
Set associative: block 12 can go anywhere in set 0 (12 MOD 4)
Block no.
0 1 2 3 4 5 6 7
Cache
Set 0
Set 1
Set 2
Set 3



This boundary moves to the right as associativity increases

For full associativity? → there is no index field (0 bits. There is just 1 set)

Exercise

- For a 40-bit memory address
- Calculate the number of bits for the tag, index, and offset fields, for the following cases:

1. Cache of 64K bytes, with blocks of 32 bytes
   a) Direct mapped, b) 2-way set associative, c) 4-way set associative

2. Cache of 1024K bytes, with blocks of 64 bytes
   a) Direct mapped, b) 4-way set associative, c) Fully associative

3. Cache of 2048K bytes, with blocks of 16 bytes
   a) Direct mapped, b) 4-way set associative, c) Fully associative

Exercise solution, Cache number 1

1: Cache of 64K bytes, with blocks of 32 bytes

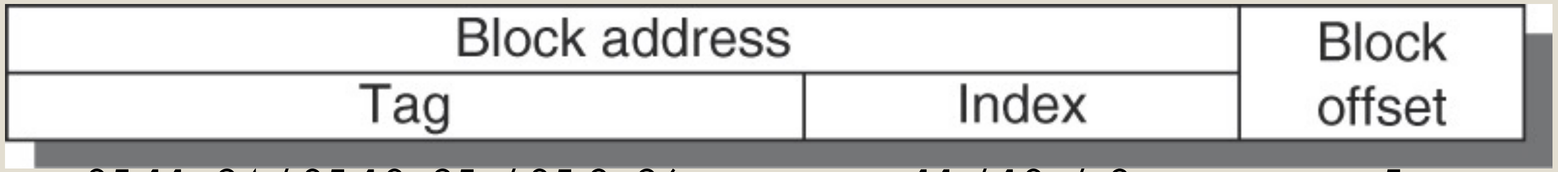
a) Direct mapped, b) 2-way set associative, c) 4-way set associative

- Blocks of 32 bytes: how many bits to select a byte (offset field) → 32 bytes: 5 bits to select a byte (32 = 2^5)
- How many blocks?

  → 64K/32 = 2K blocks
- How many sets?
  - Direct mapped: (each block is a set) → 2K blocks, 2K sets
  - 2-way set associative → 2K blocks, 2 elements per set, 1K sets
  - 4-way set associative → 2K blocks, 4 elements per set, 512 sets

Exercise solution, continuation

- How many sets?
  - Direct mapped: (each block is a set) → 2K blocks, 2K sets
  - 2-way set associative → 2K blocks, 2 elements per set, 1K sets
  - 4-way set associative → 2K blocks, 4 elements per set, 512 sets
- How many bits to select the set? (index field)
  - Direct mapped → 2K sets, 11 bits (2K=2^11)
  - 2-way set associative → 1K sets, 10 bits (1K=2^10)
  - 4-way set associative → 512 sets, 9 bits (512=2^9)



35-11=24 / 35-10=25 / 35-9=26 11 / 10 / 9 5

Exercise: solve it for Cache number 2 and 3:

- For a 40-bit memory address
- Calculate the number of bits for the tag, index, and offset fields, for the following cases:

2. Cache of 1024K bytes, with blocks of 64 bytes
   a) Direct mapped, b) 4-way set associative, c) Fully associative

3. Cache of 2048K bytes, with blocks of 16 bytes
   a) Direct mapped, b) 4-way set associative, c) Fully associative

Back to the four questions

1. Where can a block be placed (block placement)
2. How is a block found (block identification)
3. Which block should be replaced on a miss (block replacement)
4. What happens on a write (write strategy)

3: Block replacement

- Which block should be replaced on a cache miss?

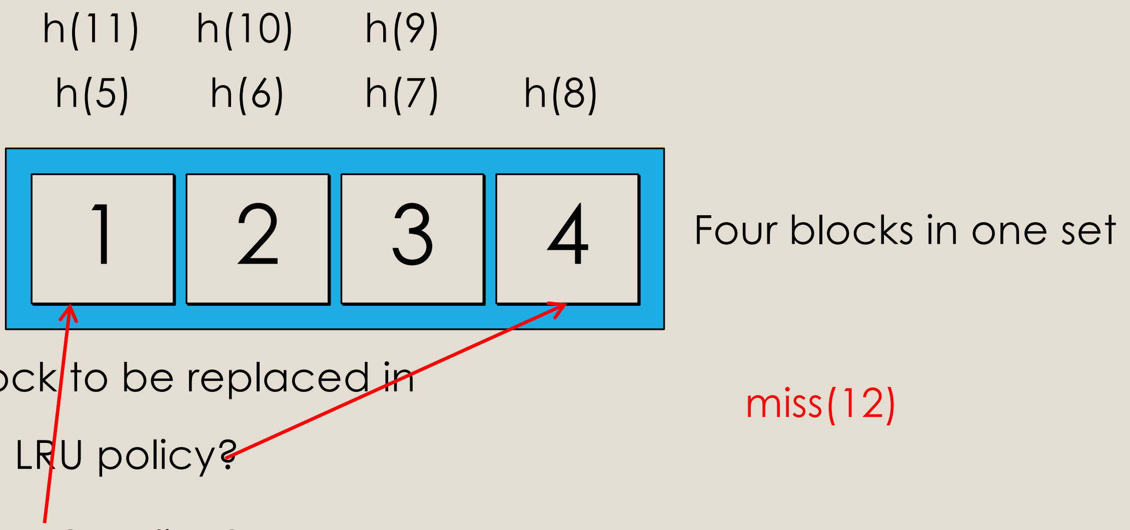
- In a direct mapped cache (1-way associative, each set is just one block)
  - only one block is checked for a miss/hit
  - if it's a miss, ***that*** *one is* the block that must be replaced

- But when associativity is higher than 1, there is a choice on which block to be evicted/replaced

Replacement Policies

- Random
  - To spread allocation uniformly: randomly choose a block within the set
- Least Recently Used (LRU)
  - Accesses to blocks are recorded
  - The block replaced is the one that has been unused the longest time
- First in first out (FIFO)
  - Replace the oldest block, i.e. the block that has been in this set for the longest time

Difference between LRU and FIFO: Example



Which is the block to be replaced in

an LRU policy?

a FIFO policy?

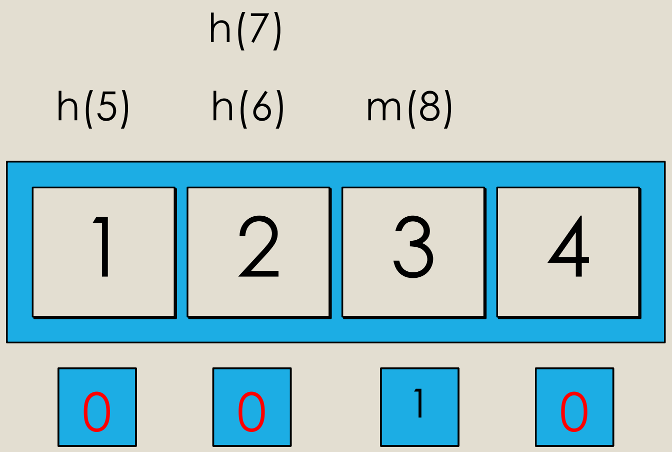
4-way associative cache

More on LRU

- Most accurate, and most expensive to implement

- Usually an approximation (pseudo LRU) is implemented when the number of ways is high:
  - One bit for each *way* (each block in a set)
  - The bit is set to 1 when the block is accessed
  - When all bits are set to 1, they are all reset to 0 but the current one

Pseudo-LRU Example



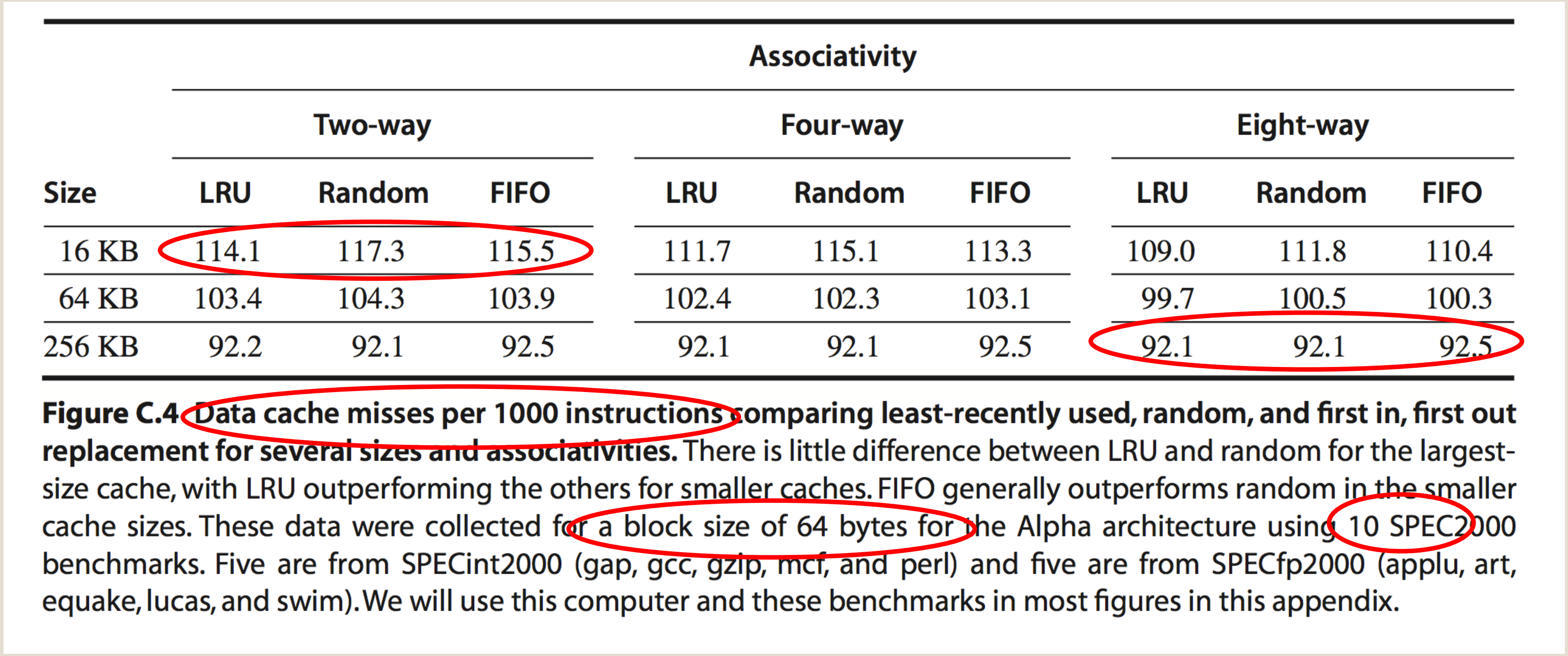
if there is a miss at time 8:
the only “0” block is chosen
for replacement
(and it is indeed the LRU one)

When all bits are 1,
they are all reset to 0,
but the current one

if there is a miss at time 9:
one of the three “0” blocks
is chosen for replacement
(usually at random)

Pseudo LRU guarantees that the block replaced is not the most RU. Sometimes it chooses indeed the LRU

Performance



	Associativity								
	Two-way			Four-way			Eight-way		
Size	LRU	Random	FIFO	LRU	Random	FIFO	LRU	Random	FIFO
16 KB	114.1	117.3	115.5	111.7	115.1	113.3	109.0	111.8	110.4
64 KB	103.4	104.3	103.9	102.4	102.3	103.1	99.7	100.5	100.3
256 KB	92.2	92.1	92.5	92.1	92.1	92.5	92.1	92.1	92.5

Figure C.4 Data cache misses per 1000 instructions comparing least-recently used, random, and first in, first out replacement for several sizes and associativities. There is little difference between LRU and random for the largest-size cache, with LRU outperforming the others for smaller caches. FIFO generally outperforms random in the smaller cache sizes. These data were collected for a block size of 64 bytes for the Alpha architecture using 10 SPEC2000 benchmarks. Five are from SPECint2000 (gap, gcc, gzip, mcf, and perl) and five are from SPECfp2000 (applu, art, equake, lucas, and swim). We will use this computer and these benchmarks in most figures in this appendix.

4: What happens on a write?

- Reads dominate processor cache accesses
  - Instruction cache: all reads
  - Data cache: average of 10% stores and 26% loads out of all instructions
- Make the common case fast → optimize for reads
- Still, writes have to be handled

Write policies

- Write - through
  - When there is a write, write on the cache and in main memory
- Write - back
  - When there is a write, write only on the cache
  - When the block is replaced, write in main memory

More on write back

- Write to main memory when a block is replaced
- BUT: we don’t want to write in main memory every time a block is replaced!
  - Only *if* it has been written

- Use a dirty bit
  - The bit is set to 1 (dirty) if a block has been written while it was in the cache
  - The bit is left at 0 (clean) if the block has not been written

- Write to main memory when a block is replaced and its dirty bit is set to 1

There are advantages in both policies

- Write – through
  - Easier to implement
  - Read misses never result in writes to the lower level
  - The levels are coherent → important in multiprocessors

- Write – back
  - Data is written less times (multiple writes within a cache block require only one write in the lower-level)

    → Less memory bandwidth and less power required

Cache Optimizations

Cache performance

Average memory access time (AMAT) =

hit time + (miss rate x miss penalty)

e.g.: if it takes 1 cycle for a hit (access to L1 cache), 20 extra cycles for
a miss (access to main memory, or next level), and we have 10% miss rate:

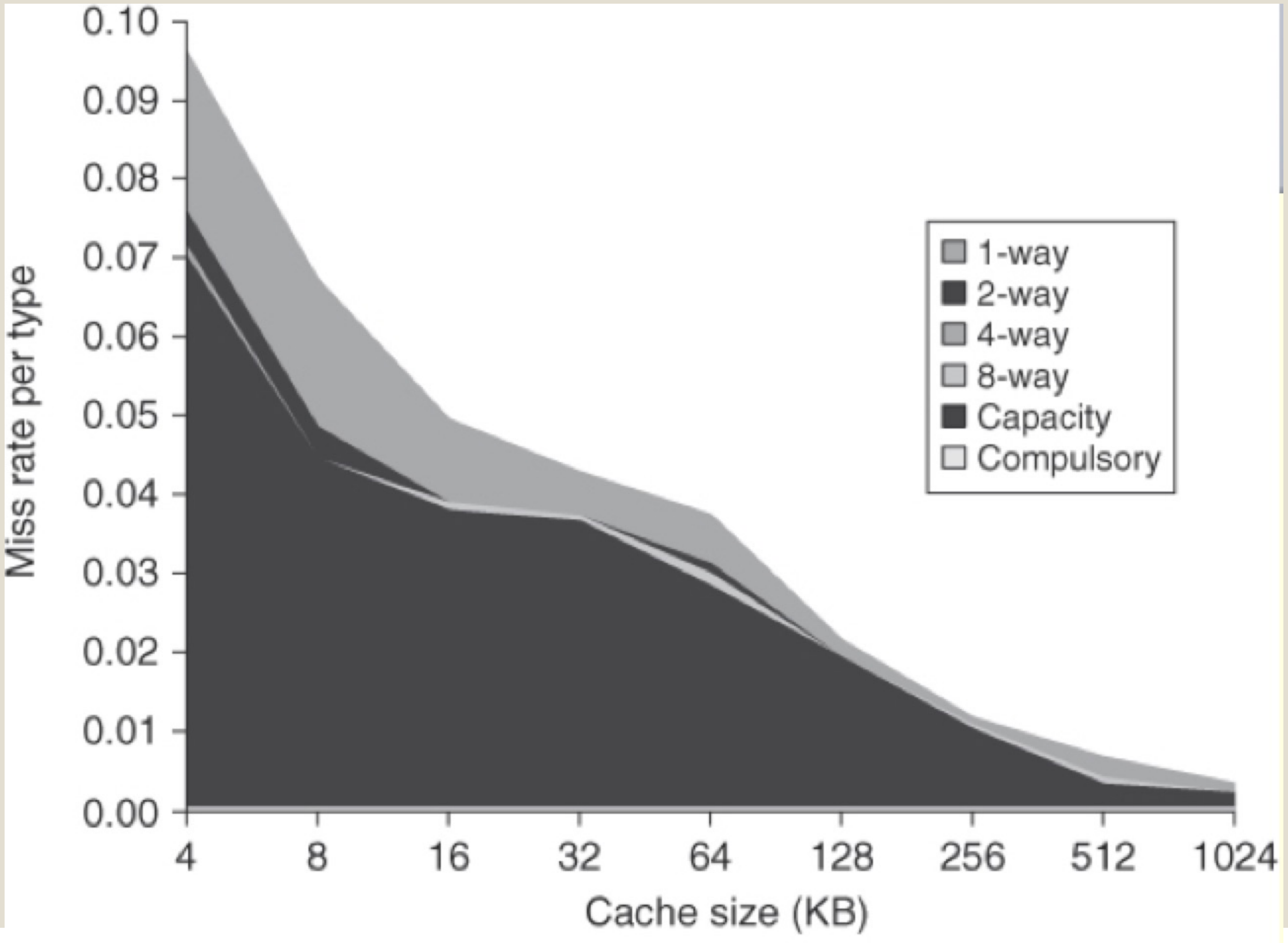
AMAT = 1 + (10% x 20) = 1 + 2 = 3

We'll see a few cache optimizations that improve (reduce) average memory access time by improving one of the three factors above

Reducing Miss Rate

- What are the causes of misses?
- A simple model helps to gain insight into it: all misses are sorted into three simple categories: (The Three Cs)
- Compulsory
  - The very first access to a block cannot be in the cache. Also called cold-start misses
- Capacity
  - If the cache cannot contain all the blocks needed during execution of a program, blocks must be discarded and later retrieved
- Conflict
  - In set associative caches a block might be discarded and later retrieved if too many blocks map to its set. Also called collision misses

Miss Rate per type



Higher Associativity reduces miss rate

But does it reduce memory access time?

Average memory access time (AMAT) =
hit time + (miss rate x miss penalty)

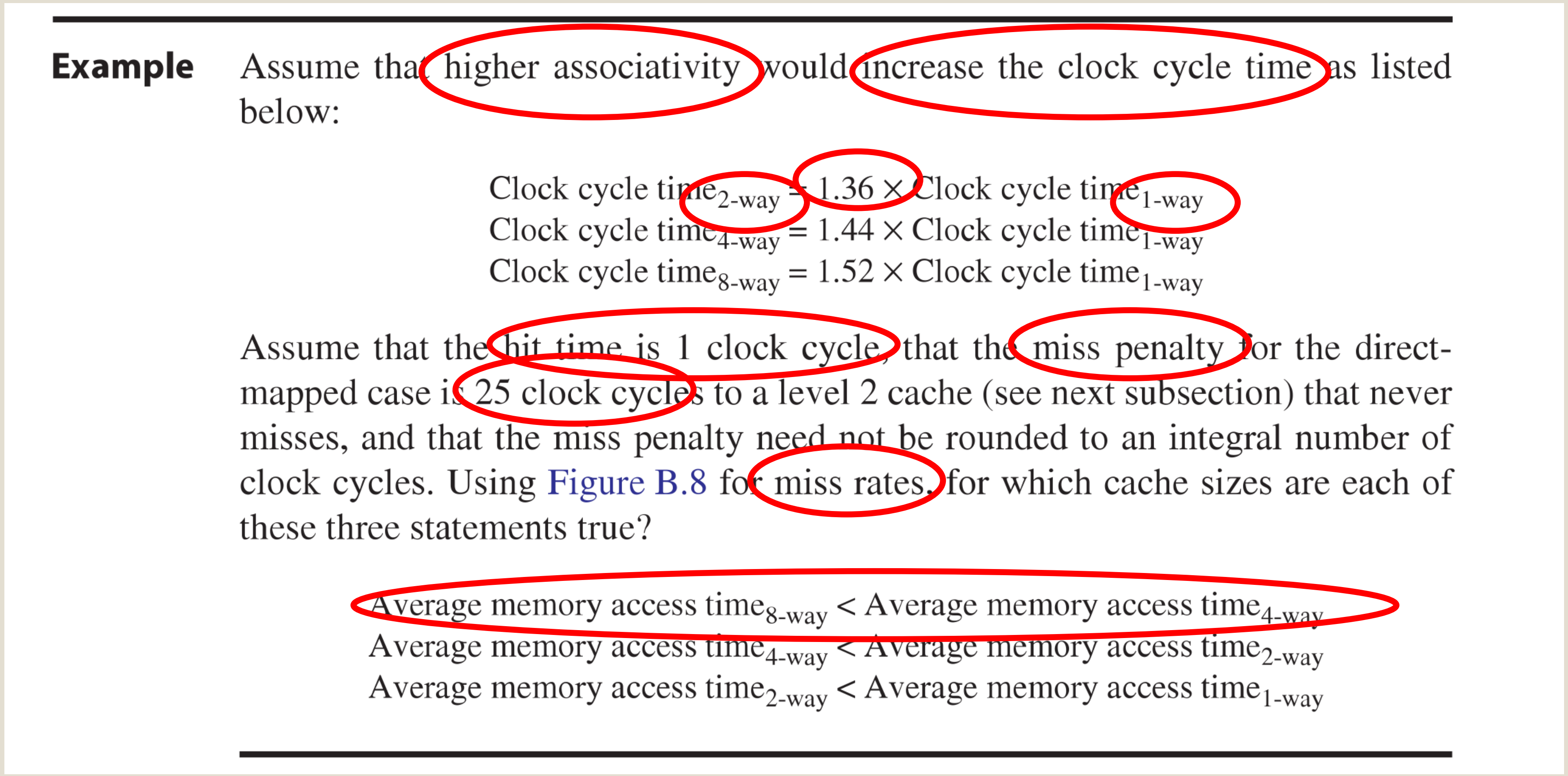
Hit Time

The critical timing path in a cache hit is a three step process

1. addressing the tag memory using the index portion of the address
2. comparing the read tag value to the address
3. setting the multiplexer to choose the correct data item if the cache is set associative

Higher associativity: more tags to be compared, larger multiplexer
→ larger hit time

Exercise from the textbook



Example Assume that higher associativity would increase the clock cycle time as listed below:

$$\text{Clock cycle time}_{\text{2-way}} = 1.36 \times \text{Clock cycle time}_{\text{1-way}}$$
$$\text{Clock cycle time}_{\text{4-way}} = 1.44 \times \text{Clock cycle time}_{\text{1-way}}$$
$$\text{Clock cycle time}_{\text{8-way}} = 1.52 \times \text{Clock cycle time}_{\text{1-way}}$$

Assume that the hit time is 1 clock cycle, that the miss penalty for the direct-mapped case is 25 clock cycles to a level 2 cache (see next subsection) that never misses, and that the miss penalty need not be rounded to an integral number of clock cycles. Using Figure B.8 for miss rates, for which cache sizes are each of these three statements true?

$$\text{Average memory access time}_{\text{8-way}} < \text{Average memory access time}_{\text{4-way}}$$
$$\text{Average memory access time}_{\text{4-way}} < \text{Average memory access time}_{\text{2-way}}$$
$$\text{Average memory access time}_{\text{2-way}} < \text{Average memory access time}_{\text{1-way}}$$

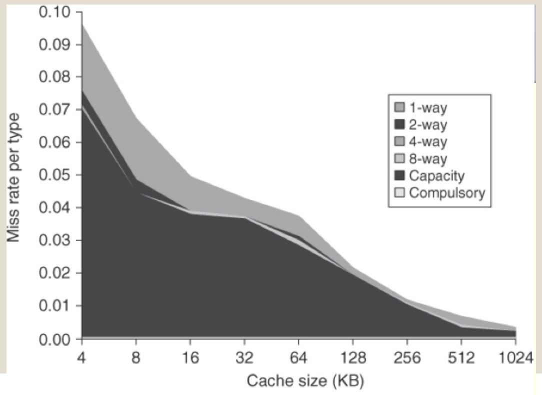
$$\text{Clock cycle time}_{2\text{-way}} = 1.36 \times \text{Clock cycle time}_{1\text{-way}}$$
$$\text{Clock cycle time}_{4\text{-way}} = 1.44 \times \text{Clock cycle time}_{1\text{-way}}$$
$$\text{Clock cycle time}_{8\text{-way}} = 1.52 \times \text{Clock cycle time}_{1\text{-way}}$$

AMAT = hit time + (miss rate x miss penalty)

$$\begin{aligned}\text{Average memory access time}_{8\text{-way}} &= \text{Hit time}_{8\text{-way}} + \text{Miss rate}_{8\text{-way}} \times \text{Miss penalty}_{8\text{-way}} \\ &= 1.52 + \text{Miss rate}_{8\text{-way}} \times 25\end{aligned}$$

$$\begin{aligned}\text{Average memory access time}_{4\text{-way}} &= 1.44 + \text{Miss rate}_{4\text{-way}} \times 25 \\ \text{Average memory access time}_{2\text{-way}} &= 1.36 + \text{Miss rate}_{2\text{-way}} \times 25 \\ \text{Average memory access time}_{1\text{-way}} &= 1.00 + \text{Miss rate}_{1\text{-way}} \times 25\end{aligned}$$

For miss rate values, we use this study (tabular form in the next slide)

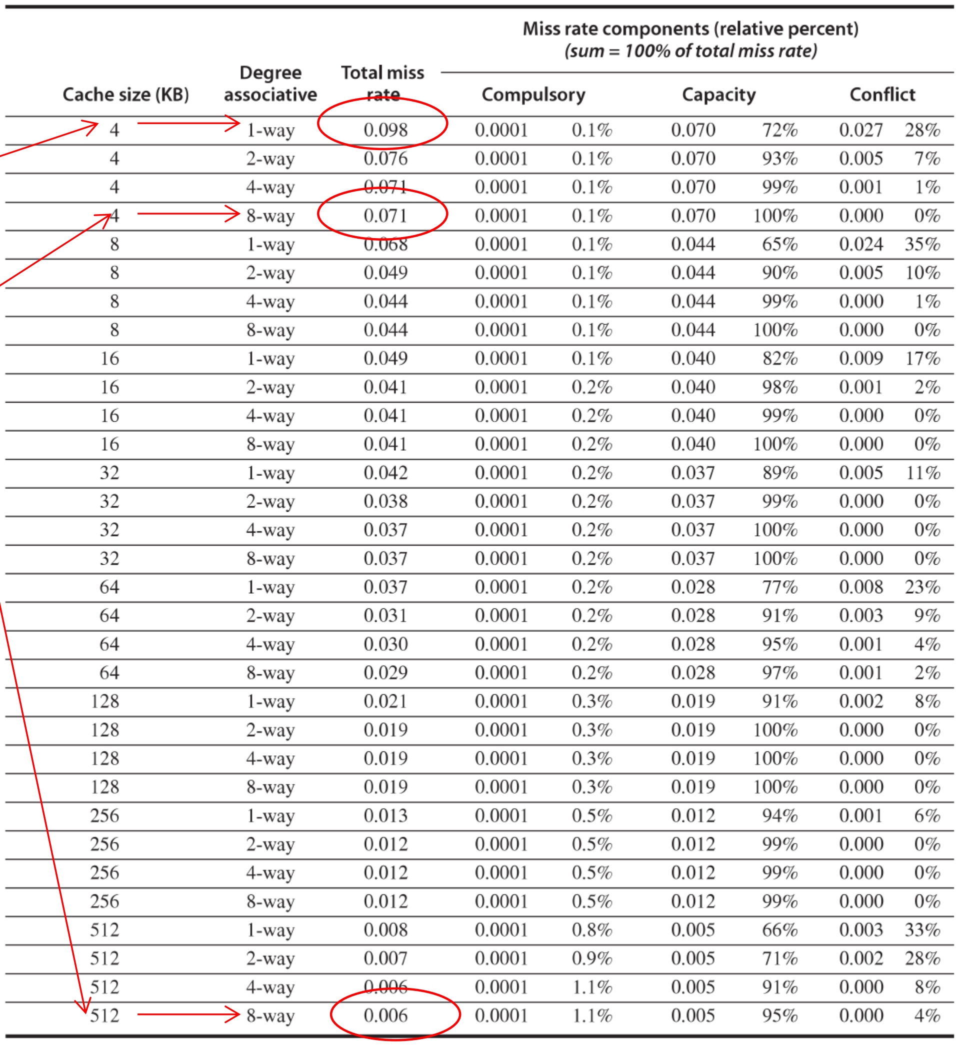


What's the miss rate for a:

4KB direct-mapped cache?

4KB 8-way set-ass. cache?

512KB 8-way set-ass. cache?



			Miss rate components (relative percent) *(sum = 100% of total miss rate)*					
Cache size (KB)	Degree associative	Total miss rate	Compulsory		Capacity		Conflict	
4	1-way	0.098	0.0001	0.1%	0.070	72%	0.027	28%
4	2-way	0.076	0.0001	0.1%	0.070	93%	0.005	7%
4	4-way	0.071	0.0001	0.1%	0.070	99%	0.001	1%
4	8-way	0.071	0.0001	0.1%	0.070	100%	0.000	0%
8	1-way	0.068	0.0001	0.1%	0.044	65%	0.024	35%
8	2-way	0.049	0.0001	0.1%	0.044	90%	0.005	10%
8	4-way	0.044	0.0001	0.1%	0.044	99%	0.000	1%
8	8-way	0.044	0.0001	0.1%	0.044	100%	0.000	0%
16	1-way	0.049	0.0001	0.1%	0.040	82%	0.009	17%
16	2-way	0.041	0.0001	0.2%	0.040	98%	0.001	2%
16	4-way	0.041	0.0001	0.2%	0.040	99%	0.000	0%
16	8-way	0.041	0.0001	0.2%	0.040	100%	0.000	0%
32	1-way	0.042	0.0001	0.2%	0.037	89%	0.005	11%
32	2-way	0.038	0.0001	0.2%	0.037	99%	0.000	0%
32	4-way	0.037	0.0001	0.2%	0.037	100%	0.000	0%
32	8-way	0.037	0.0001	0.2%	0.037	100%	0.000	0%
64	1-way	0.037	0.0001	0.2%	0.028	77%	0.008	23%
64	2-way	0.031	0.0001	0.2%	0.028	91%	0.003	9%
64	4-way	0.030	0.0001	0.2%	0.028	95%	0.001	4%
64	8-way	0.029	0.0001	0.2%	0.028	97%	0.001	2%
128	1-way	0.021	0.0001	0.3%	0.019	91%	0.002	8%
128	2-way	0.019	0.0001	0.3%	0.019	100%	0.000	0%
128	4-way	0.019	0.0001	0.3%	0.019	100%	0.000	0%
128	8-way	0.019	0.0001	0.3%	0.019	100%	0.000	0%
256	1-way	0.013	0.0001	0.5%	0.012	94%	0.001	6%
256	2-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%
256	4-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%
256	8-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%
512	1-way	0.008	0.0001	0.8%	0.005	66%	0.003	33%
512	2-way	0.007	0.0001	0.9%	0.005	71%	0.002	28%
512	4-way	0.006	0.0001	1.1%	0.005	91%	0.000	8%
512	8-way	0.006	0.0001	1.1%	0.005	95%	0.000	4%

Figure B.8 Total miss rate for each size cache and percentage of each according to the three C's. Compulsory misses are independent of cache size, while capacity misses decrease as capacity increases, and conflict misses decrease as associativity increases. Figure B.9 shows the same information graphically. Note that a direct-mapped cache of size *N* has about the same miss rate as a two-way set-associative cache of size *N*/2 up through 128 K. Caches larger than 128 KB do not prove that rule. Note that the Capacity column is also the fully associative miss rate. Data were collected as in Figure B.4 using LRU replacement.

Back to the exercise:

$$\text{Average memory access time}_{8\text{-way}} = \text{Hit time}_{8\text{-way}} + \text{Miss rate}_{8\text{-way}} \times \text{Miss penalty}_{8\text{-way}}$$

$$= 1.52 + \text{Miss rate}_{8\text{-way}} \times 25$$

$$\begin{aligned}
\text{Average memory access time}_{4\text{-way}} &= 1.44 + \text{Miss rate}_{4\text{-way}} \times 25\\
\text{Average memory access time}_{2\text{-way}} &= 1.36 + \text{Miss rate}_{2\text{-way}} \times 25\\
\text{Average memory access time}_{1\text{-way}} &= 1.00 + \text{Miss rate}_{1\text{-way}} \times 25
\end{aligned}$$

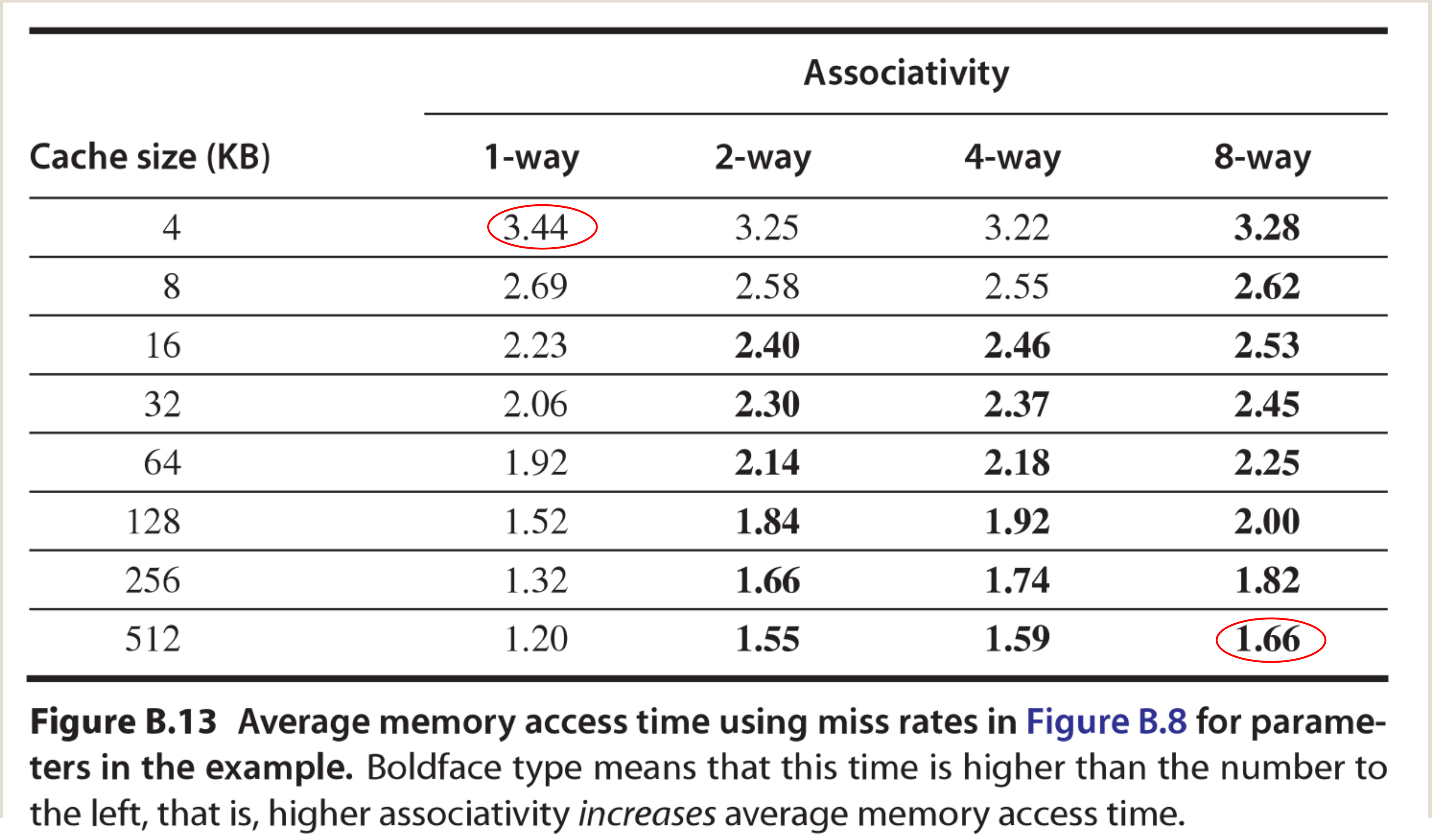
AMAT for a 4KB direct-mapped cache:

$$\text{Average memory access time}_{1\text{-way}} = 1.00 + (0.098 \times 25) = 3.44$$

AMAT for a 512 KB 8-way set associative cache:

$$\text{Average memory access time}_{8\text{-way}} = 1.52 + (0.006 \times 25) = 1.66$$

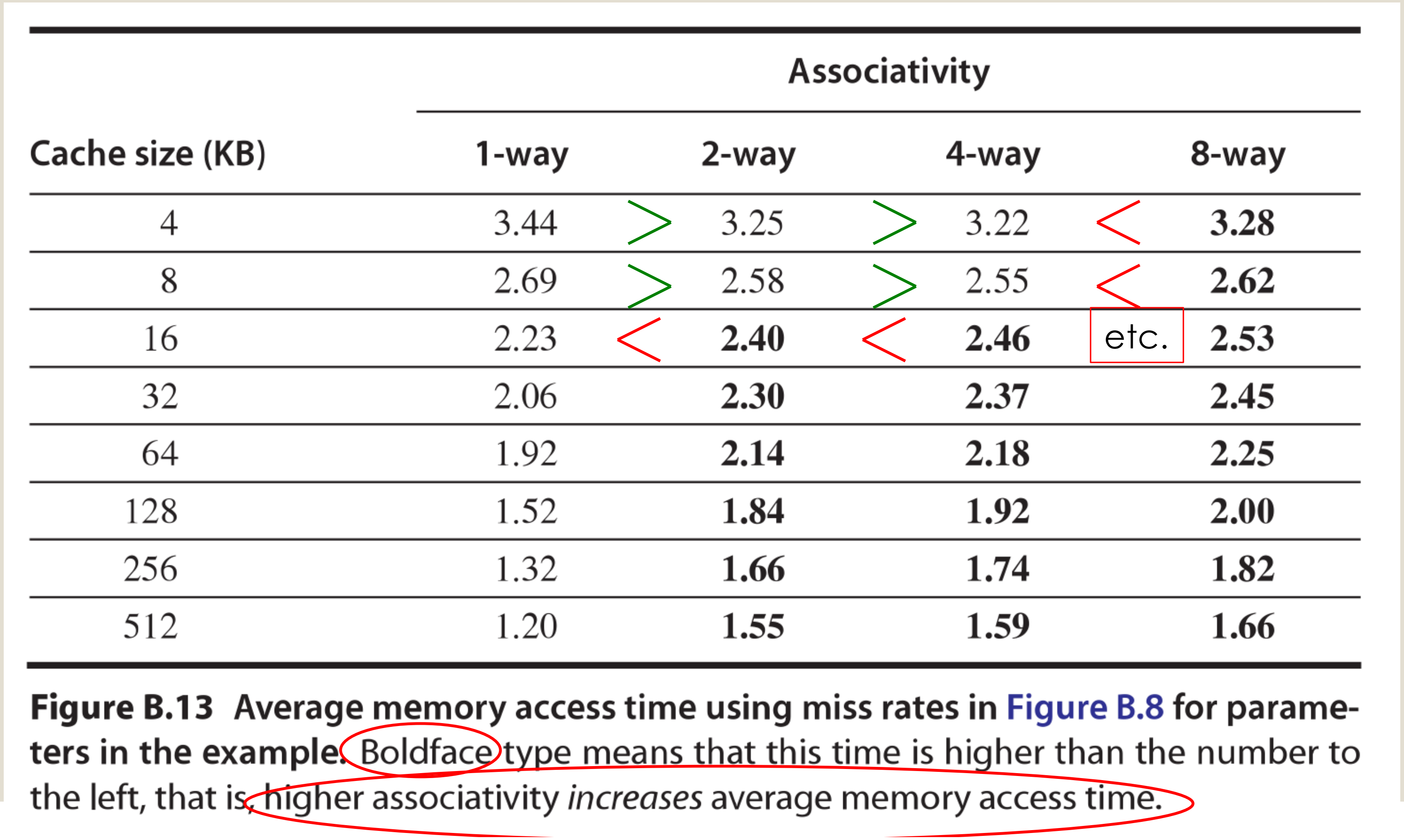
If we map all numbers and combinations:



	Associativity			
Cache size (KB)	**1-way**	**2-way**	**4-way**	**8-way**
4	3.44	3.25	3.22	**3.28**
8	2.69	2.58	2.55	**2.62**
16	2.23	**2.40**	**2.46**	**2.53**
32	2.06	**2.30**	**2.37**	**2.45**
64	1.92	**2.14**	**2.18**	**2.25**
128	1.52	**1.84**	**1.92**	**2.00**
256	1.32	**1.66**	**1.74**	**1.82**
512	1.20	**1.55**	**1.59**	**1.66**

Figure B.13 Average memory access time using miss rates in Figure B.8 for parameters in the example. Boldface type means that this time is higher than the number to the left, that is, higher associativity *increases* average memory access time.

Original question: does higher associativity decrease average memory access time?



	Associativity						
Cache size (KB)	1-way		2-way		4-way		8-way
4	3.44	>	3.25	>	3.22	<	**3.28**
8	2.69	>	2.58	>	2.55	<	**2.62**
16	2.23	<	**2.40**	<	**2.46**	etc.	**2.53**
32	2.06		**2.30**		**2.37**		**2.45**
64	1.92		**2.14**		**2.18**		**2.25**
128	1.52		**1.84**		**1.92**		**2.00**
256	1.32		**1.66**		**1.74**		**1.82**
512	1.20		**1.55**		**1.59**		**1.66**

Figure B.13 Average memory access time using miss rates in Figure B.8 for parameters in the example. Boldface type means that this time is higher than the number to the left, that is, higher associativity *increases* average memory access time.

Different parameter values → different conclusions

- for example, if miss penalty is higher (for example 250 cycles instead of 25) then the numbers of the previous table shift:
  - higher associativity would decrease AMAT more often

Take home point

- higher associativity may decrease AMAT or may increase AMAT → it depends on various factors, as seen in the previous exercise
- In general, the three variables in the AMAT formula are affected in different ways by any given optimization
- e.g. higher associativity *increases* hit time, *decreases* miss rate

Various cache optimizations

1. Multilevel Caches: Small and Simple First-Level Caches to reduce Hit Time and Power
2. Way Prediction to reduce Hit Time
3. Pipelined Cache Access to increase Cache Bandwidth
4. Nonblocking Caches to increase Cache Bandwidth
5. Compiler Optimizations to reduce Miss Rate

1: Multilevel Caches

Should the cache be:

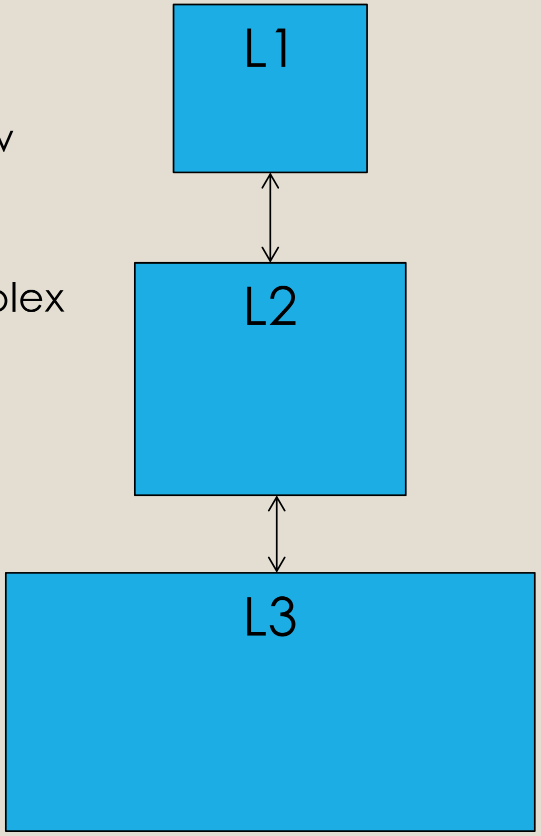
1. Larger and with high associativity, to reduce miss-rate, or:
2. Smaller and simple (low associativity) to reduce hit time

do both! With multilevel caches

Multilevel Caches

Small and simple
first level cache
to keep the hit time low

Larger and more complex
second level cache
to avoid the cost of
going to main memory
too many times



Multilevel caches are
also energy efficient
(the larger the cache,
the more power needed
to access it)

Energy consumption as a function of associativity and size

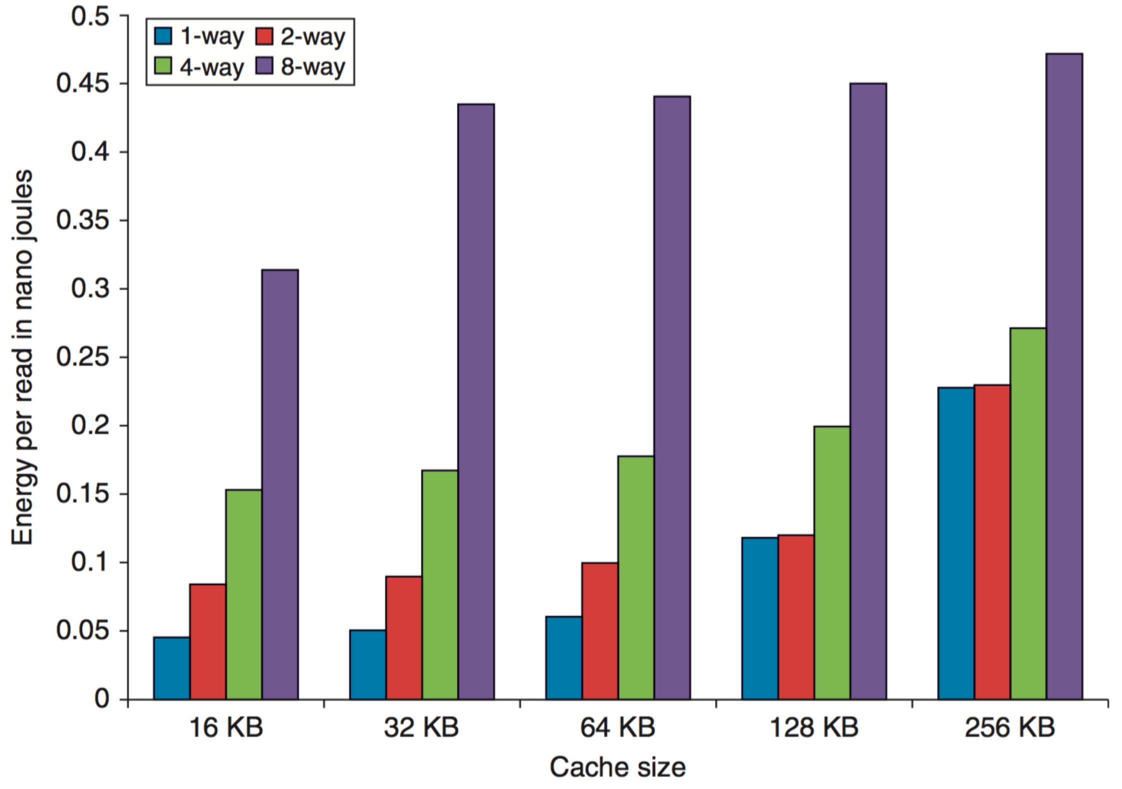


Figure 2.4 Energy consumption per read increases as cache size and associativity are increased. As in the previous figure, CACTI is used for the modeling with the same technology parameters. The large penalty for eight-way set associative caches is due to the cost of reading out eight tags and the corresponding data in parallel.

- ARM Cortex-A8
  - First level cache 16KB to 32KB, 4-way-associative
  - Second level cache 128KB to 1MB, 8-way associative

- Intel i7
  - First level cache 32KB, 4-way (I) and 8-way (D)
  - Second level cache 256 KB, 8-way
  - Third level cache 2MB, 16-way

2: Way Prediction to reduce Hit Time

- Extra bits are kept in the cache to predict which way (out of an n-way associative cache) will be likely to be the right one (just like branch prediction)
- Only a single tag comparison is made, and the multiplexer is set early to choose the predicted block → small hit time
- The corresponding data is read in parallel with the comparison (as in direct mapped caches)
- If the prediction is wrong, the correct data is read in the next cycle(s) (and the prediction bit is changed)

2: Way Prediction numbers

- Prediction accuracy of 90% for a two-way cache
- Prediction accuracy of 80% for a four-way cache
- Prediction accuracy higher for I than for D cache

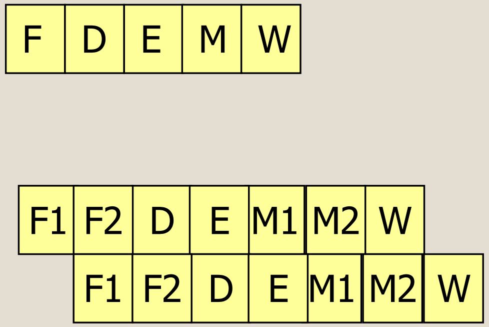
- Used by the ARM-Cortex-A8 (4way associative)

3 Pipelined Cache Access
to increase cache bandwidth

- If the processor frequency is so high that first level cache access time doesn't fit in one cycle:
  - pipeline the access to first level cache
  - So that you can still serve one memory instruction per cycle (if it's a hit) even though the access latency is higher

Example of first level cache access times:

Intel Pentium 1 (mid 1990): 1 cycle
Intel Pentium 3 (2000): 2 cycles
Intel Core i7 (2018): 4 cycles



4: Nonblocking Caches
to increase cache bandwidth

- If the processor allows OOO execution and we have a cache miss:
- We want even memory instructions that follow the miss (i.e. that are later in the code) to be able to go ahead

- This can be done with nonblocking caches: caches that are designed to keep serving requests even when serving a miss
  - hit under miss, or even hit under multiple miss

4: Nonblocking Caches
to increase cache bandwidth

- Both Intel I7 and ARM Cortex A8 have nonblocking caches
  - ARM Cortex A8 can serve hits-under-miss
  - Intel I7 even hits-under-multiple-misses

- So far we have seen only HW techniques to improve AMAT
- Can things be done, to improve AMAT, even before run-time?

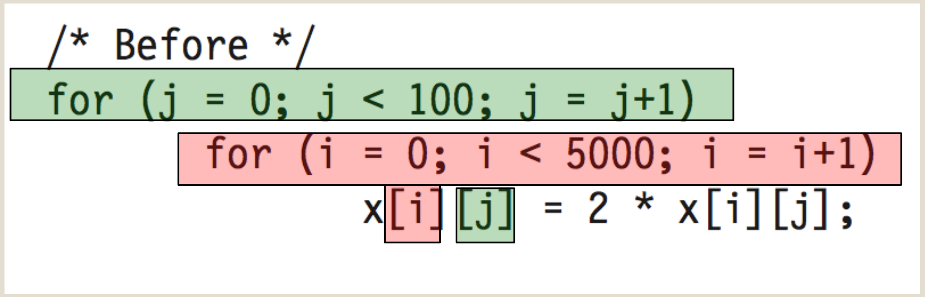
5: Compiler Optimizations
to reduce miss rate

- Not only Hardware techniques:
- Compiler techniques can also help using the cache better

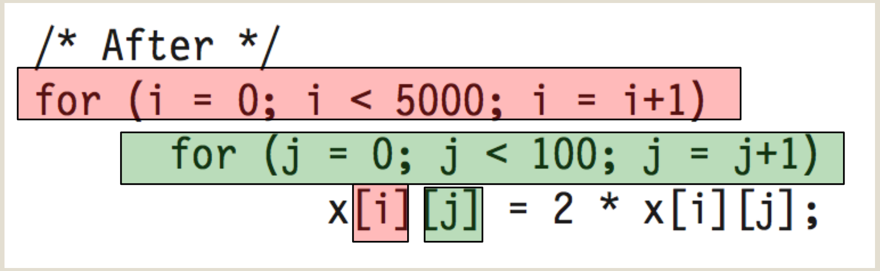
Interchange and Blocking

- Loop Interchange
  - Swap nested loops to access memory in sequential order
- Loop Blocking
  - Instead of accessing entire rows or columns, subdivide matrices into blocks
  - Improves locality of accesses

Loop Interchange

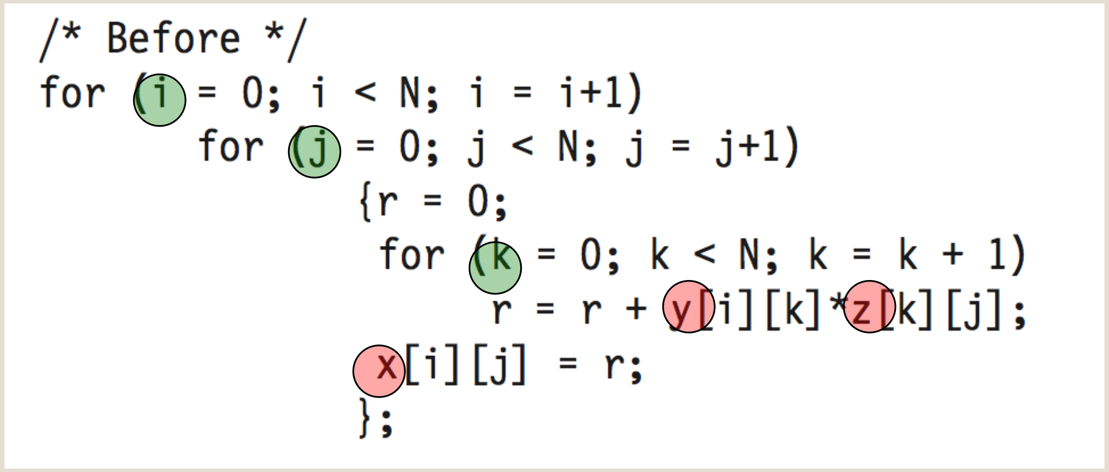


```
/* Before */
for (j = 0; j < 100; j = j+1)
        for (i = 0; i < 5000; i = i+1)
                x[i][j] = 2 * x[i][j];
```



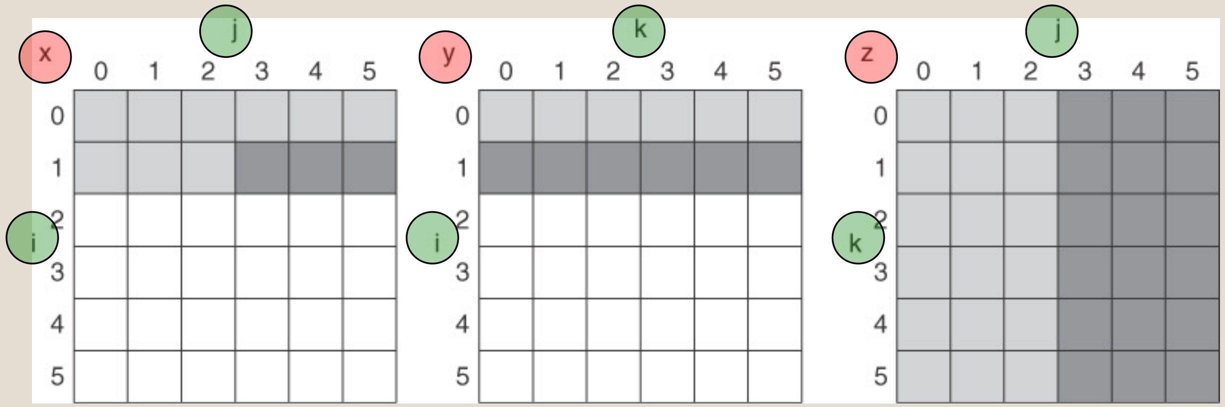
```
/* After */
for (i = 0; i < 5000; i = i+1)
        for (j = 0; j < 100; j = j+1)
                x[i][j] = 2 * x[i][j];
```

Loop Blocking



```
/* Before */
for (i = 0; i < N; i = i+1)
        for (j = 0; j < N; j = j+1)
                {r = 0;
                 for (k = 0; k < N; k = k + 1)
                      r = r + y[i][k]*z[k][j];
                 x[i][j] = r;
                };
```

A snapshot of the three arrays x, y, and z when *N* = 6 and i = 1.



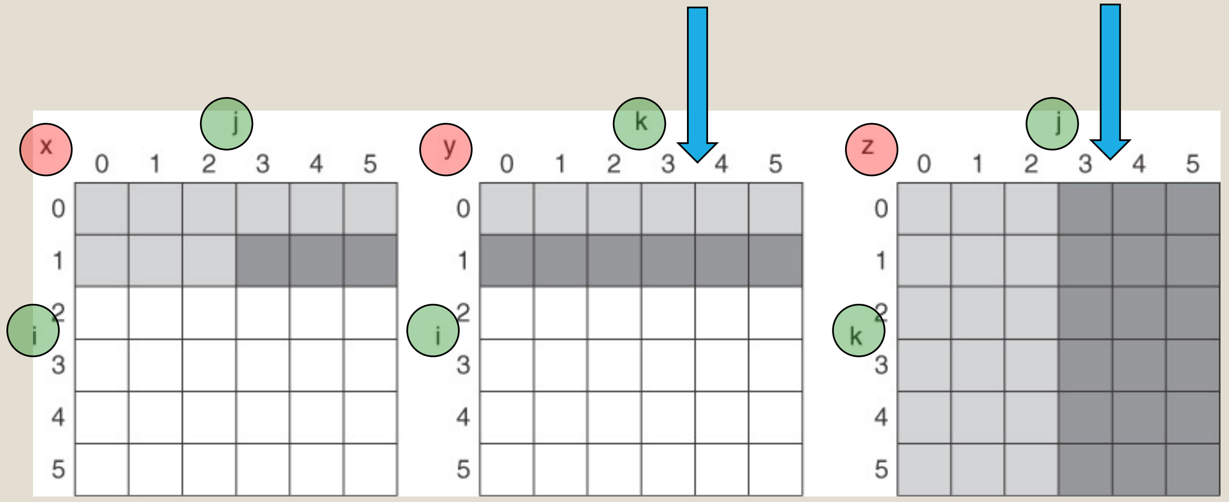
The age of accesses to the array elements is indicated by shade:
white means not yet touched,
light means older accesses,
and dark means newer accesses.

Elements of y and z are read repeatedly to calculate new elements of x.

Loop Blocking

If the cache can hold ALL NxN elements, all is well (provided there are no conflicts)

If N is large and the cache cannot hold ALL NxN elements, elements (which are still needed) will continuously be evicted



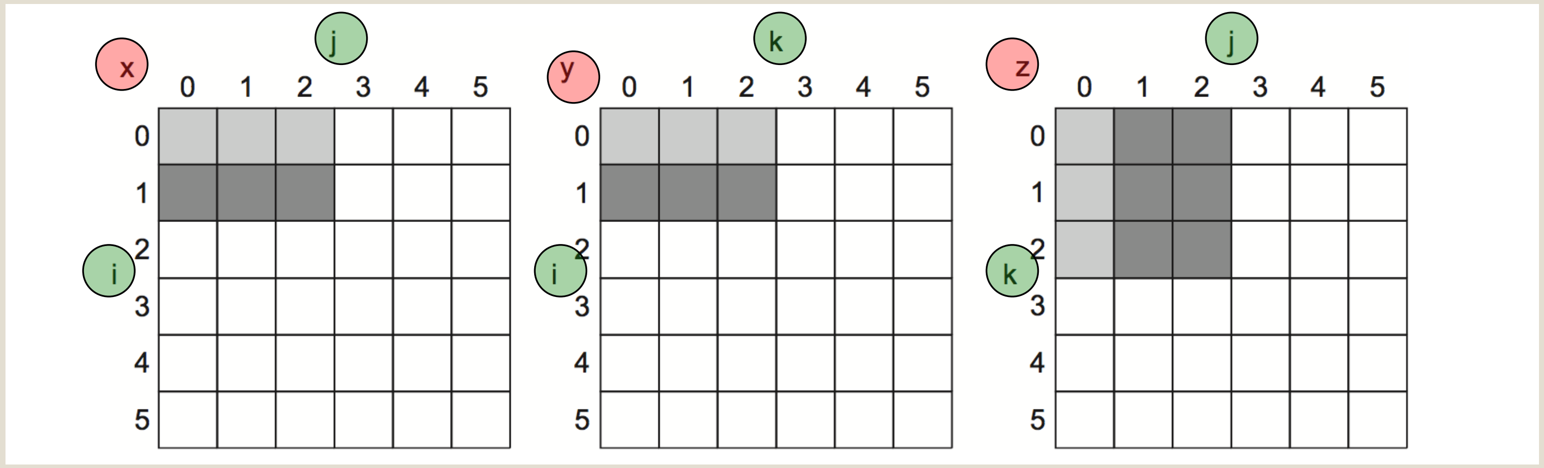
```
/* After */
for (jj = 0; jj < N; jj = jj+B)
for (kk = 0; kk < N; kk = kk+B)
for (i = 0; i < N; i = i+1)
       for (j = jj; j < min(jj+B,N); j = j+1)
           {r = 0;
            for (k = kk; k < min(kk+B,N); k = k + 1)
                 r = r + y[i][k]*z[k][j];
            x[i][j] = x[i][j] + r;
           };
```

the original code is changed to compute on a submatrix of size B by B

Two inner loops now compute in steps of size B rather than the full length (N) of x and z

B is called the *blocking factor*

New access pattern



If BxB elements fit in the cache, now as they are read over and over again there are no misses