Temporal locality → item itself

Spatial locality → nearby addresses

Direct Mapped Cache Simulation (E=1)

$M = 16$ byte addresses $\quad 2^4 \Rightarrow 4$ bit

$B = 2$ bytes / block $\Rightarrow 2^1 = 1 = |BO|$

$S = 4$ sets $= 2^2 \Rightarrow |S| = 2$

$E = 1$ Blocks / set

PA

Tag	SI		BO

	V	Tag	Block	
00	1	1	M[8]	M[9]
01	0			
10	0			
11	1	0	M[6]	M[7]

0	0000	miss
1	0001	hit
7	0111	miss
8	1000	miss
0	0000	miss

Cache size $= C = S \times E \times B$

For E-way set associative case, we need to compare all tags on that set with our tag

↳ no match: one line is selected for eviction and replacement

2-way set Associative Cache Simulation (E=2)

$M = 16$ byte addresses $= 2^4 \Rightarrow |PA| = 4$

$B = 2$ bytes / block $= 2^1 \Rightarrow |BO| = 1$

$S = 2$ sets $= 2^1 \Rightarrow |SI| = 1$

$|Tag| = 2$

Tag		SI	BO

$E = 2$ Blocks / set

	SI	
0	0000	miss
1	0001	hit
7	0111	miss
8	1000	mis
0	0000	hit

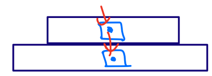
	V	Tag	Block		V	Tag	Block	
0	1	00	M[0]	M[1]	1	10	M[8]	M[9]
1	1	01	M[6]	M[7]				

Write operations

What to do on a write-hit? → The line you want to write is on the cache.

Write-through (write immediately to the memory)

Write both in cache and to memory



Write-back (defer write to mem until replacement of line)

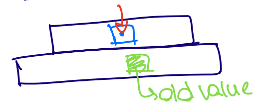
Need a dirty bit (line different from mem or not)

D	V	Tag	Data

D=0 → no need to update the mem

D=1 →

What to do on a write-miss? → You want to write into a mem address which is not in the cache.



Write-allocate

Load into cache, update line in cache

→ when we have to replace the block, update data back to mem if the write bit is set.

→ otherwise, don't.

No-write-allocate

Writes immediately to the memory

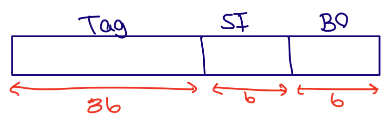
Typical

Write through + no-write-allocate

Write-back + write allocate

Örnek: L1 i-cache, d-cache

32 KB, 8 way, Mem Address = 48 bit



$C = 32\ KB = 2^5 \times 2^{10} = 2^{15}$

$E = 8 = 2^3$ → 8 lines in each set

$S = 2^6$ $\qquad |SI| = 6$

$C = 2^{15} = S \times E \times B \Rightarrow \dfrac{2^{15}}{2^3 \times 2^6} = 2^6 = B$ $\qquad |BO| = 6$

TYPES OF CACHE MISSES

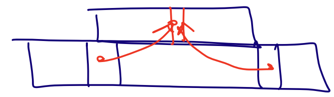
Cold (compulsory) miss

The cache is empty

Capacity miss

The set of active cache blocks (working set) is larges than the cache.

Conflict miss → Most likely to happen in direct-mapped access



Multiple data objects all map to the same level k block

CACHE PERFORMANCE METRICS

Miss rate

Fraction of memory references not found in cache (misses / accesses)

= 1 − hit rate

Hit time

Time to deliver a line in the cache to the processor

↳ Includes time to determine whether the line is in the cache

Miss Penalty

Additional time required because of a miss

Hit rate vs. Miss rate

Assume { cache hit time → 1 cycle; miss penalty → 100 cycles }

% 97 hits : 1 cycle + 0.03 * 100 cycles = 4 cycles

% 99 hits : 1 cycle + 0.01 * 100 cycles = 2 cycles

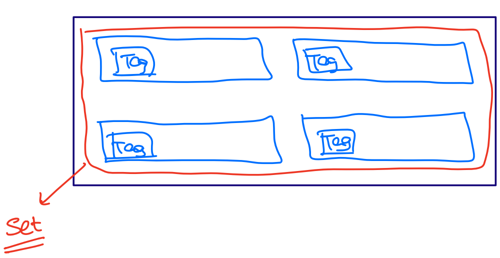
~~hit rate % 97 is twice as good as % 99~~

↳ use miss rate

Given $C = S \times E \times B$

Max number of Tag comparisons needed

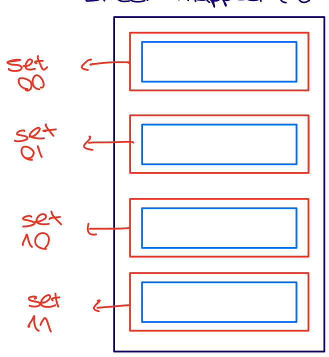
Fully Associative $(S=1)$



Address	Tag	BO

1-tag comparison needed

Direct-mapped $(E=1)$

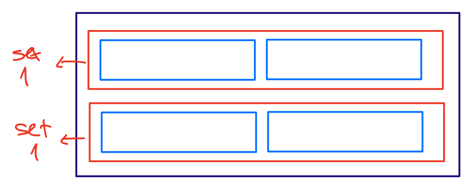


Tag	SI	BO

2

$(E=2)$

2-way Set Associative



Tag	SI	BO

1

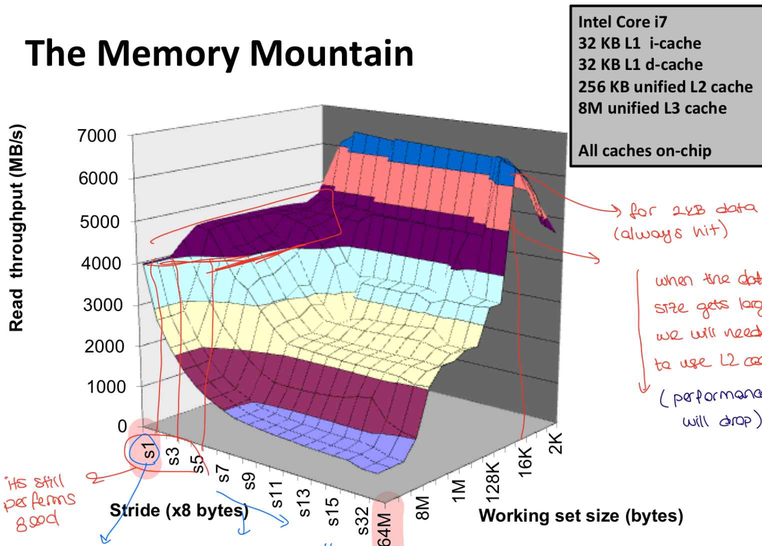
THE MEMORY MOUNTAIN

Read throughput

Number of bytes read from mem per second (MB/s)

Memory Mountain

Measured read throughput as a function of spatial and temporal locality



for 2KB data (always hit)

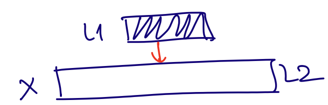
when the data size gets larger we will need to use L2 caches (performance will drop)

it's still performs good

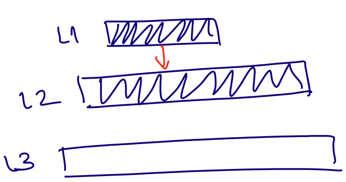
1 miss, 7 hits... 1 miss 4 hits

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 47

working set (WS) < 32 KB



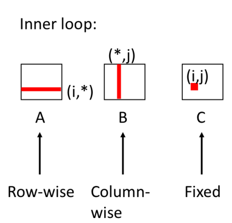
32 KB < WS < 256 KB



Matrix Multiplication (ijk)

2 loads, 0 stores
misses/iter = 1.25

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```



Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

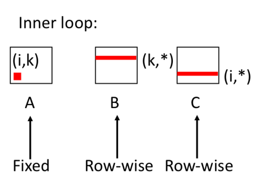
→ I'm accessing this 1 time (negligible)

$|a_{ij}| = 8$ bytes

Block size = 32 bytes

2 loads, 1 store
misses/iter = 0.5

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```



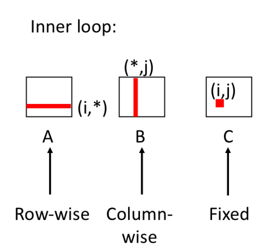
Access only 1 time

Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (jik)

```
/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum
  }
}
```

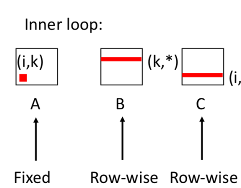


Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (ikj)

```
/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

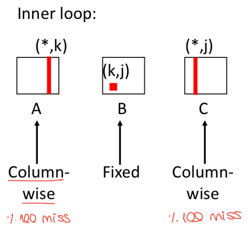


Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (jki)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

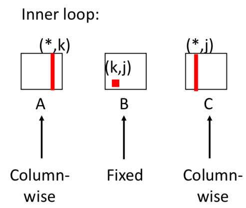


Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Matrix Multiplication (kji)

```
/* kji */
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```



Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0