CENG 3420
Computer Organization & Design
Lecture 13: Cache

Textbook: Chapter 5.3-5.4

Zhengrong Wang

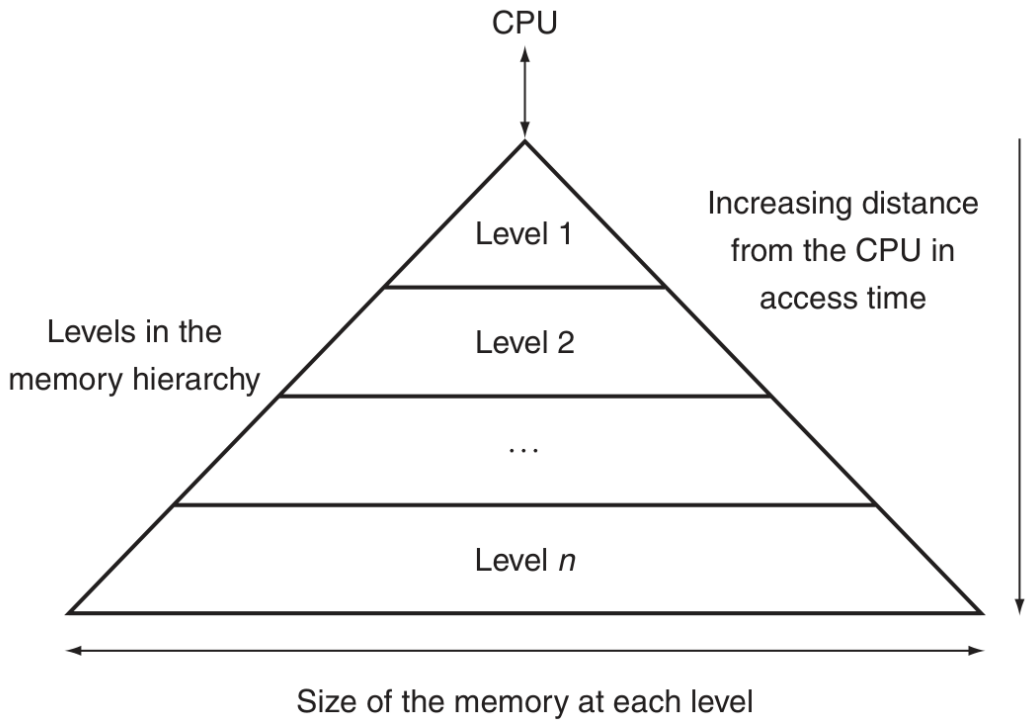
CSE Department, CUHK

zhengrongwang@cuhk.edu.hk

Introduction

Cache to Exploit Locality

- Small but fast caches help exploit locality:
  - Register file (1-2KiB, 1 cycle).
  - L1 cache (64-128KiB, 1-4 cycles).
  - L2 cache (256-512KiB, 10 cycles).
  - L3 cache (1-2MiB, 50 cycles).
  - Main memory (16-32GiB, 100 cycles).
  - SSD (256GiB-1TiB, 10^5 cycles).

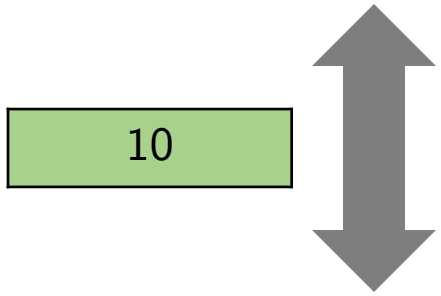


General Cache Concept

Cache

4	9	10	3

Smaller, faster, more expensive memory caches (keeps) a subset of memory



Data is copied in block-sized transfer units.

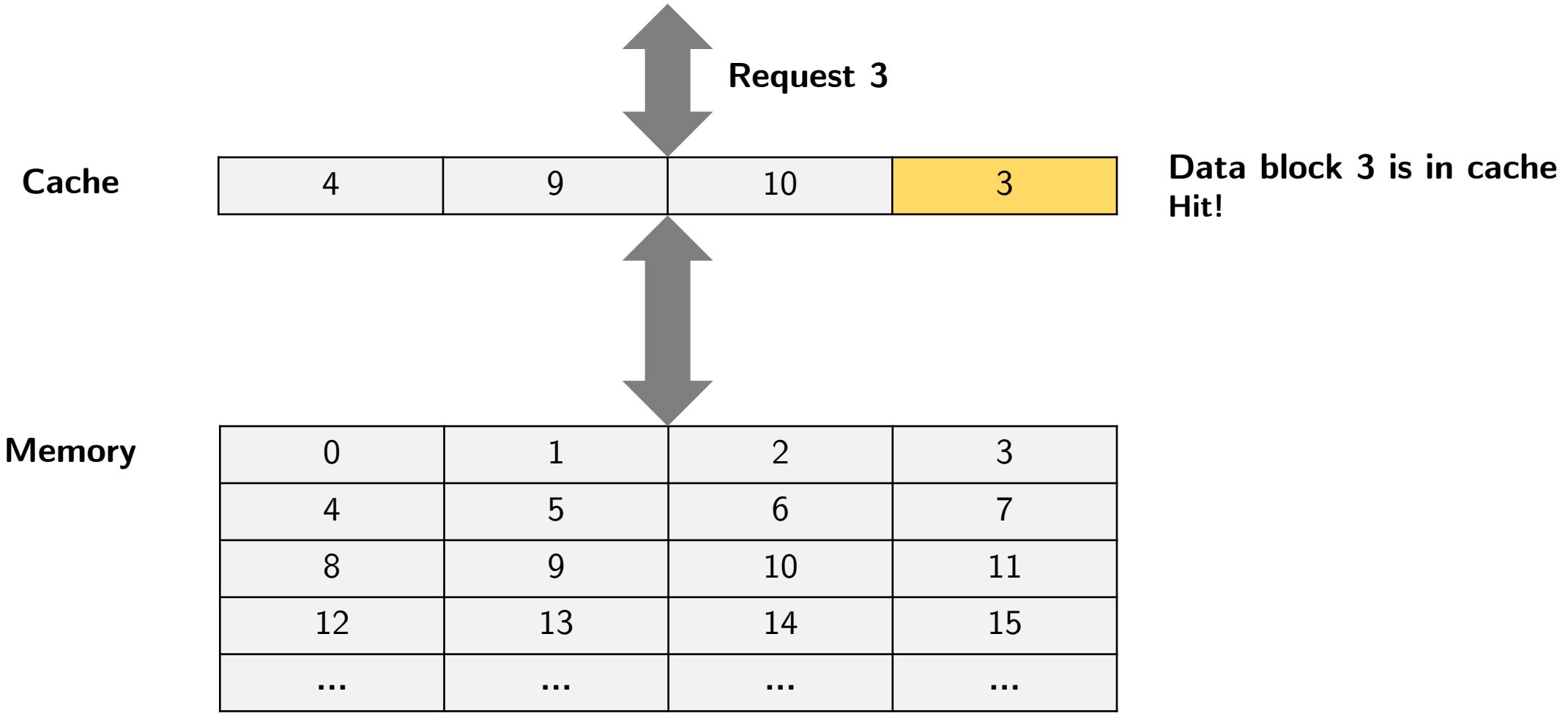
Memory

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
...	...	...	...

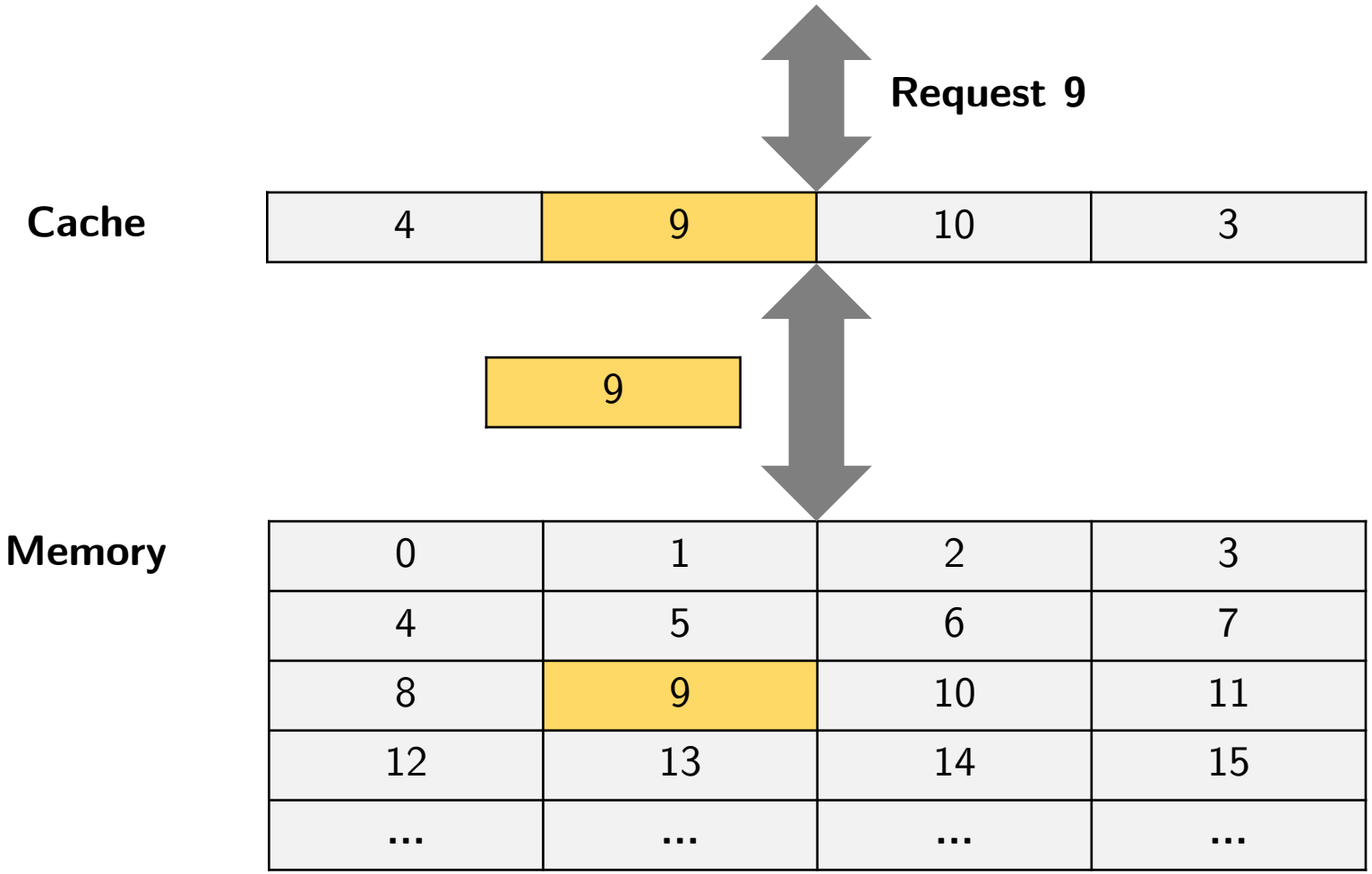
Larger, slower, cheaper memory

viewed partitioned into “blocks”

General Cache Concept: Hit



General Cache Concept: Miss



Data block 9 is not in cache.
Miss!

Fetch block 9 from memory.

Block 9 is placed in cache.

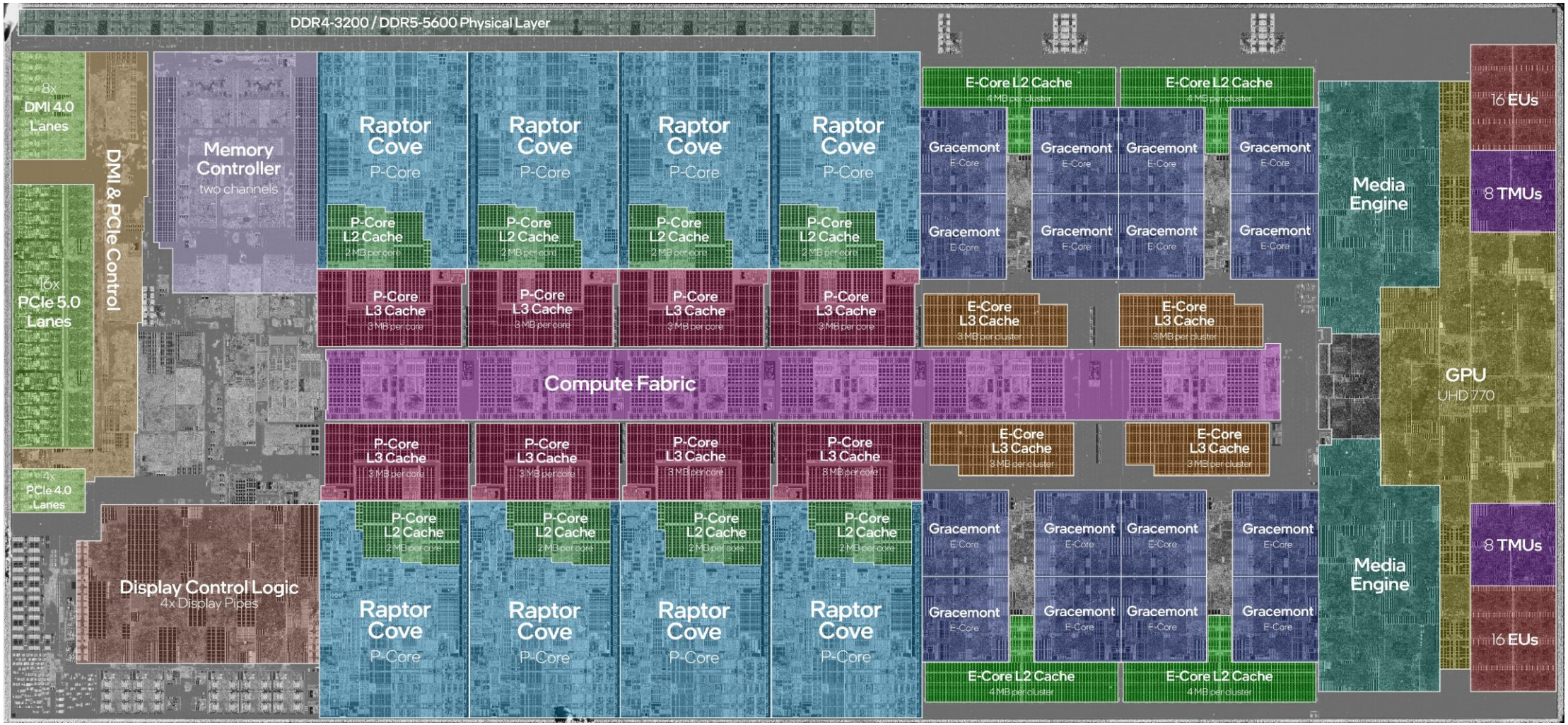
- ***Placement policy:***

Determine where block goes.

- ***Replacement policy:***

Determine which block (victim) to evict to make space for new block.

Intel Core i9 13900K Die Photo



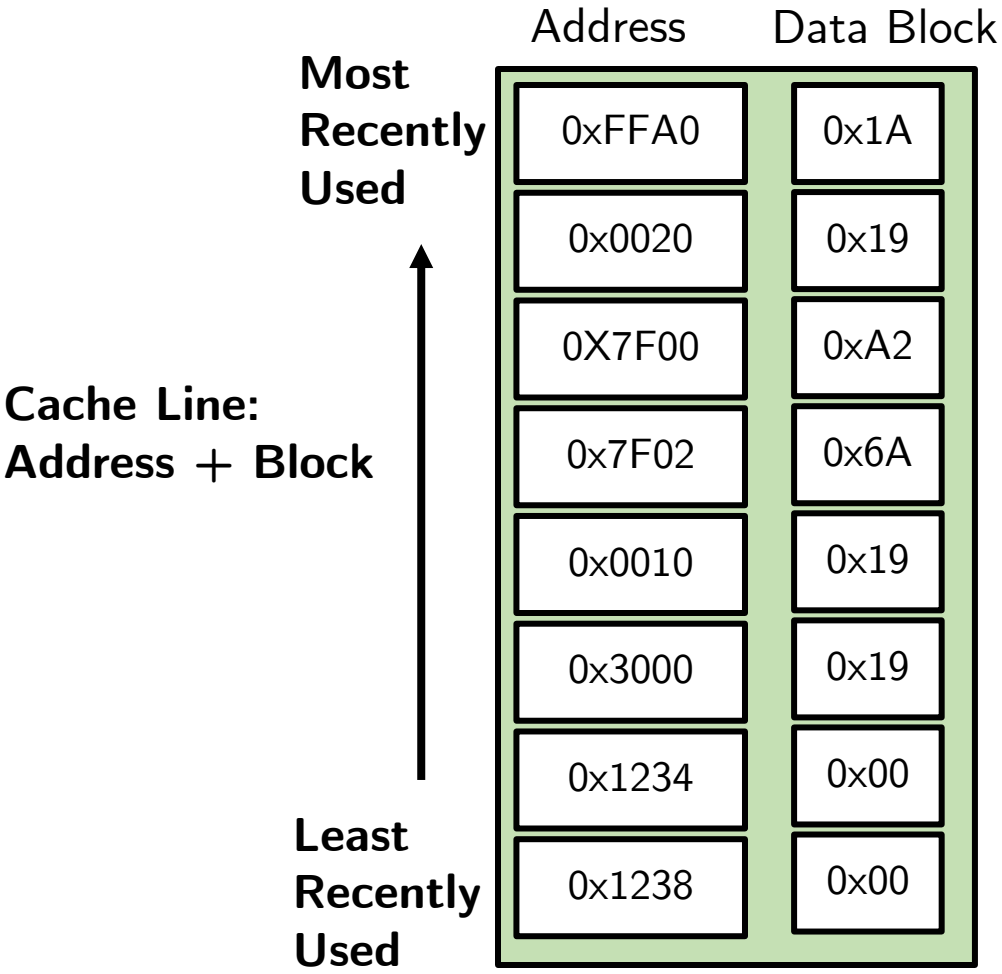
Designing a Cache

Designing a Cache

- Main Questions:
  - How do I do a cache block lookup?
  - How/when do I insert a new block?
  - How do I choose a block to put back?

- Design Goals:
  - Simple and fast.
  - Store most useful data.

Conceptual Design 1: Fully Associative Cache



- How to lookup data?
  - Check all addresses, see if one matches.
- What to evict on cache fill?
  - Replace the least recently used (known as LRU), on bottom of picture
  - Insert new item at top
- The Good: Very flexible
- The Bad: Complex hardware for large caches!

Assumptions: address size is 0x16 bit, block size is 8-bit (1 byte), byte addressable memory

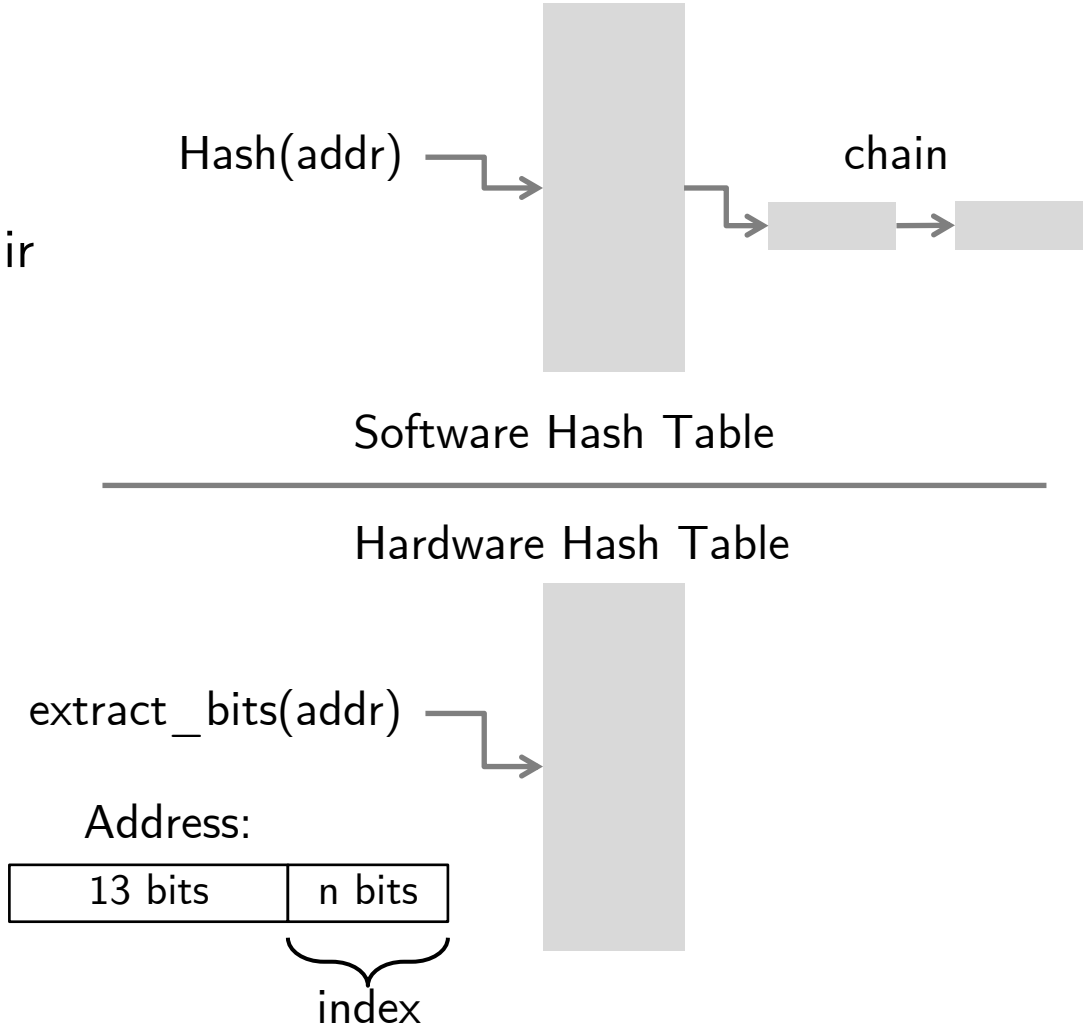
Can we make it simper?

Insight:

- Don't need full flexibility to map any address anywhere – we also don't need to check all locations at once ... it's overkill...
- Can we restrict how much of the cache we need to check?
- We can use the address to limit the scope of placement!

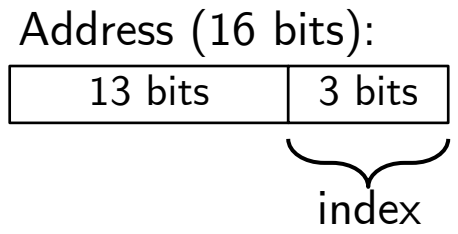
Alternative Data Structure

- What about a hash table?
  - Hash address to find a data value
  - Each entry of the hash table stores (addr, data) pair
- Problem 1: Real hash functions take too long
  - Just use some bits of the address
- Problem 2: Accessing chain takes too long
  - Insight: we don't need to cache all elements
  - Just drop elements that don't fit...
  - Maybe just lose a little performance



Conceptual Design 2: Direct Mapped

Last 3 bits end in:	Address	Data Block
0:	0xFFA0	0x1A
1:	0x0029	0x19
2:	0X7F02	0xA2
3:	0x7F03	0x6A
4:	0x0034	0x19
5:	0x301D	0x19
6:	0x1216	0x00
7:	0x121F	0x00

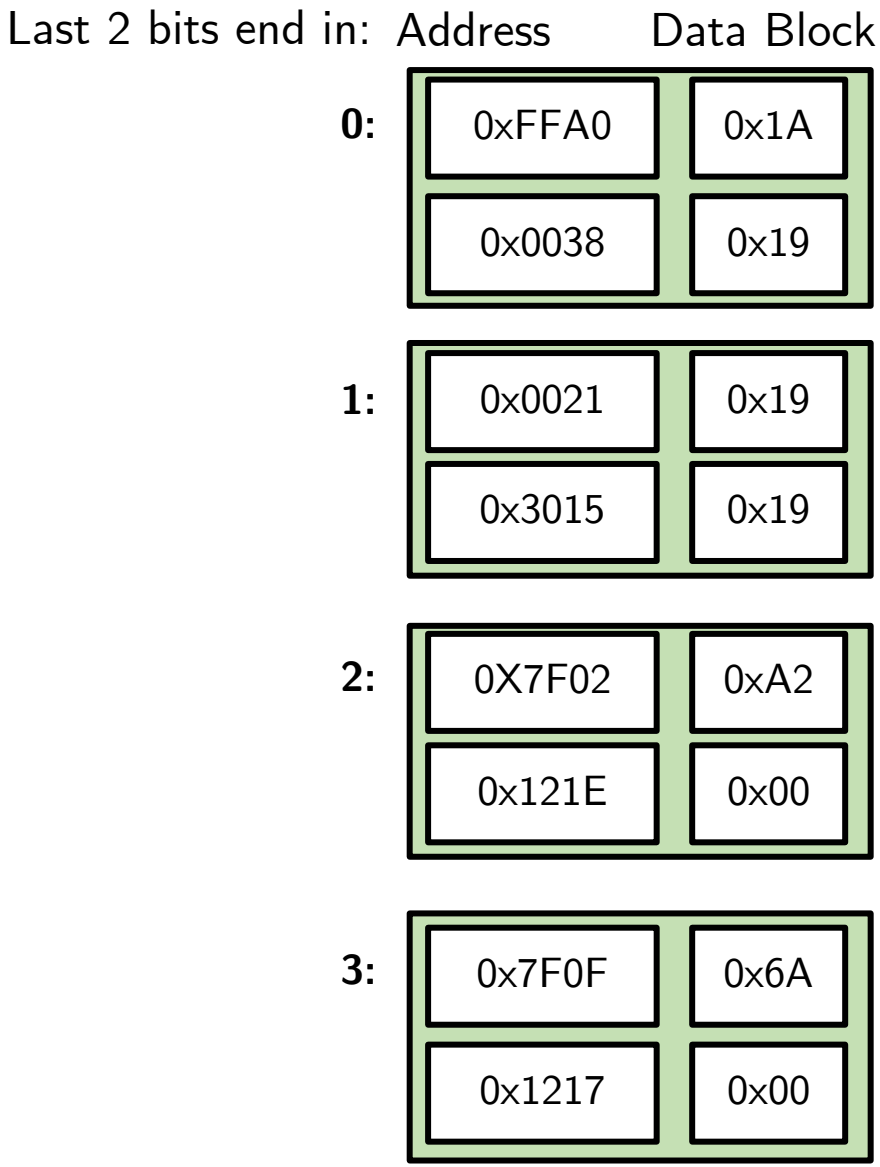


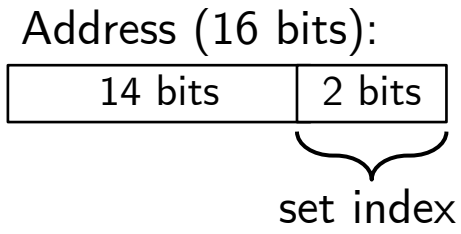
- How to lookup data?
  - Look at the lower bits of the address to determine which element to check
- What to evict when we insert?
  - Only evict the element which the address maps to.
- The good: Less work for hardware to perform.
- The bad: Sometimes we evict a useful item.

General Cache Design

Fully Associative	**Set Associative**	**Direct Mapped**
Complex, Good Replacement Policy	Associative within a smaller subset of the data.	Simple, Sometimes Bad Replacement

Conceptual Design 3: Set Associative

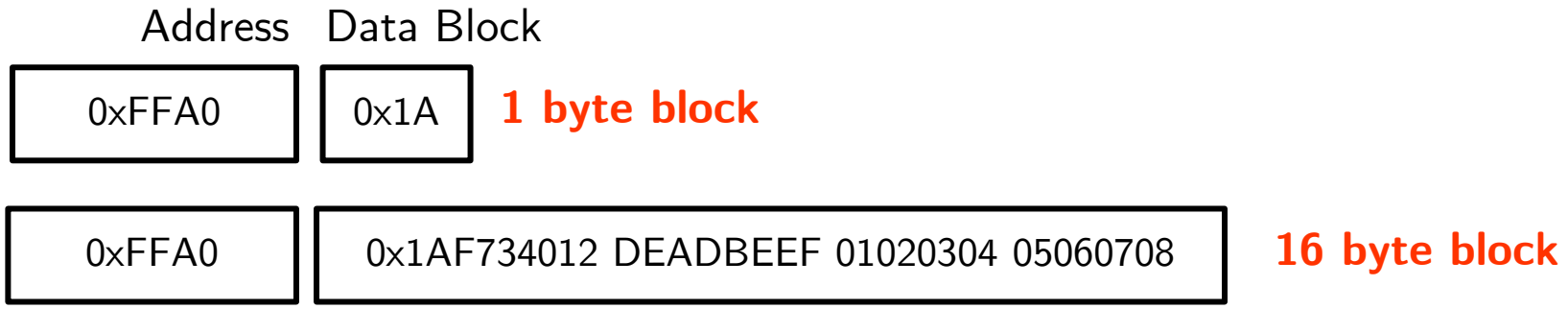




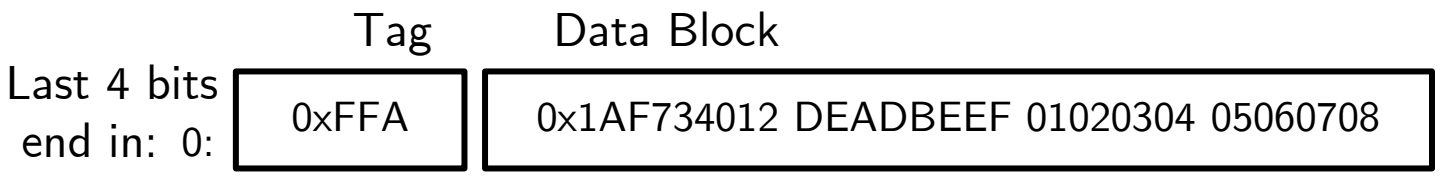
- How to lookup data?
  - Look at the lower bits of the address to determine which set to check
  - Then check all the addresses in that set
- What to evict on cache insertion?
  - Evict one of the two options.
  - E.g., the least recently used.
- Tradeoff between complexity and flexibility

Address Overhead

- Problem: Addresses take up too much overhead
- Solution 1: Bigger cache blocks: 1 byte -> 16 bytes (or more!)



- Solution 2: Don't store the whole address!

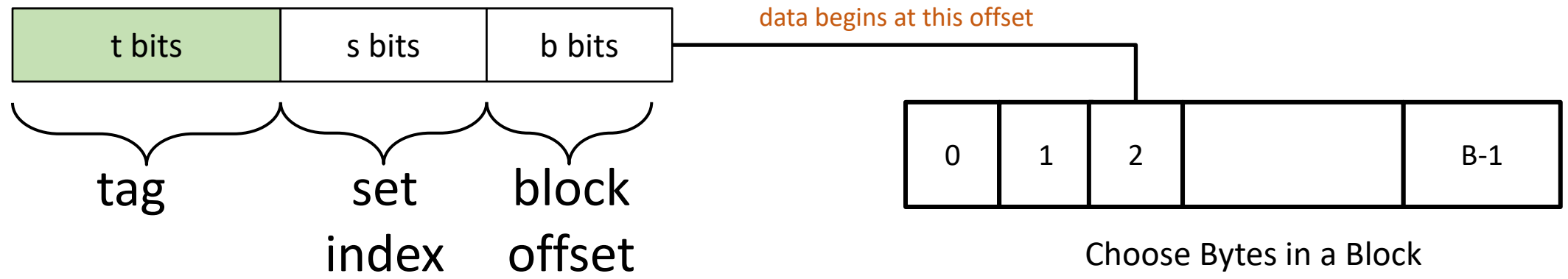


  - Tag: Truncated address

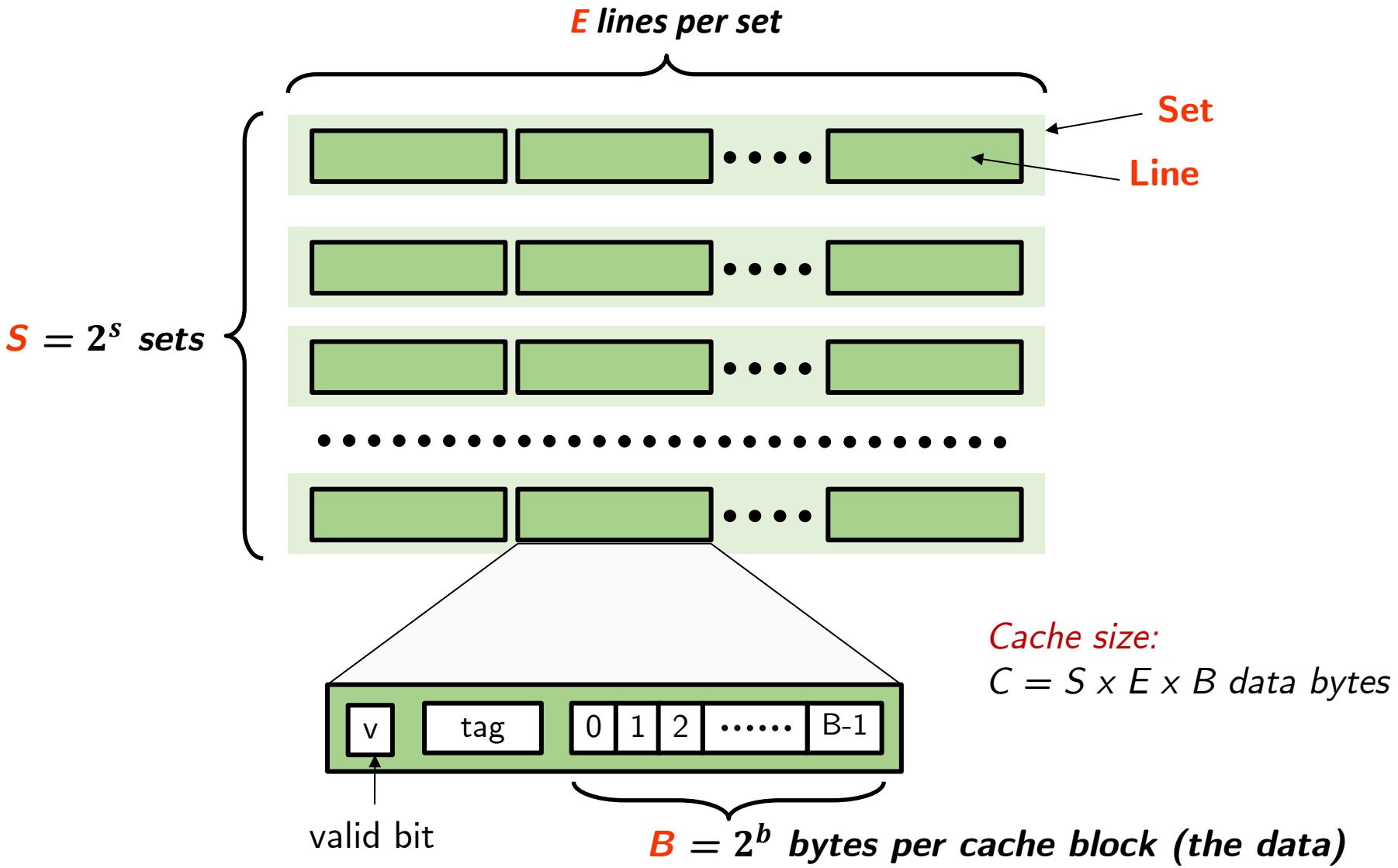
How to Index into a Cache

- Some bits are for the bytes within a block
- Some bits are for the set
- Some bits are for the tag

Address of word:

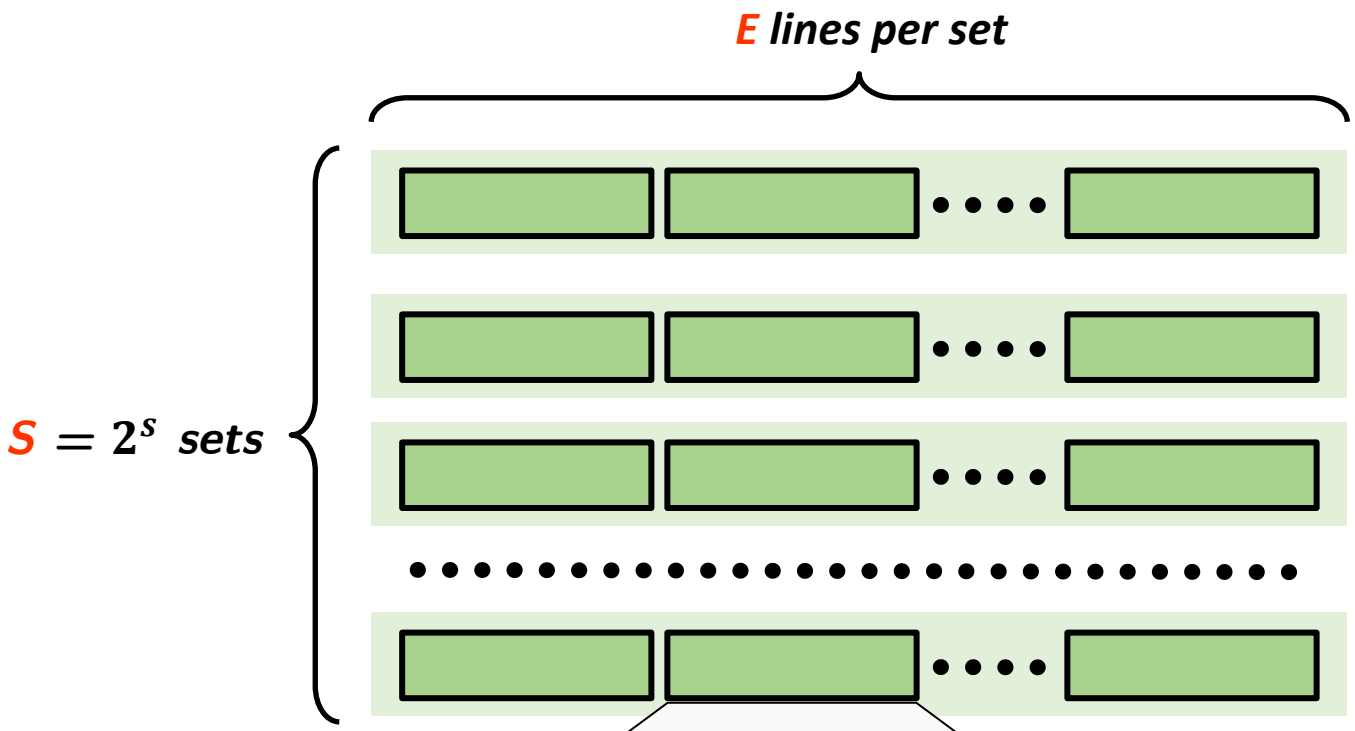


General Set Associative Cache (S, E, B)



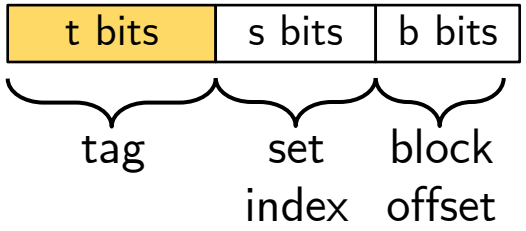
Cache size:
C = S x E x B data bytes

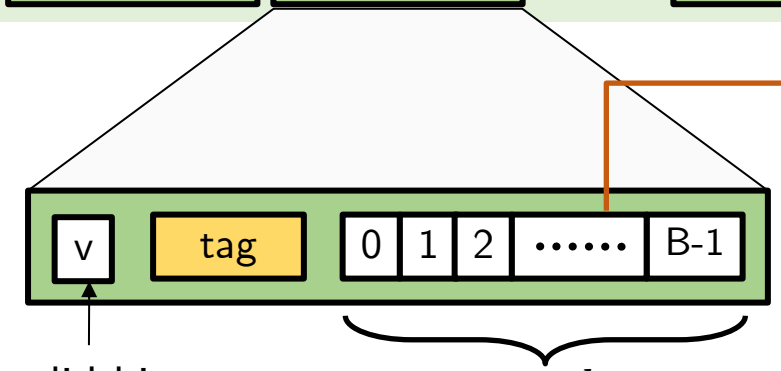
Cache Read



- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid → hit*
- *Return data starting at that offset*

Address of word:



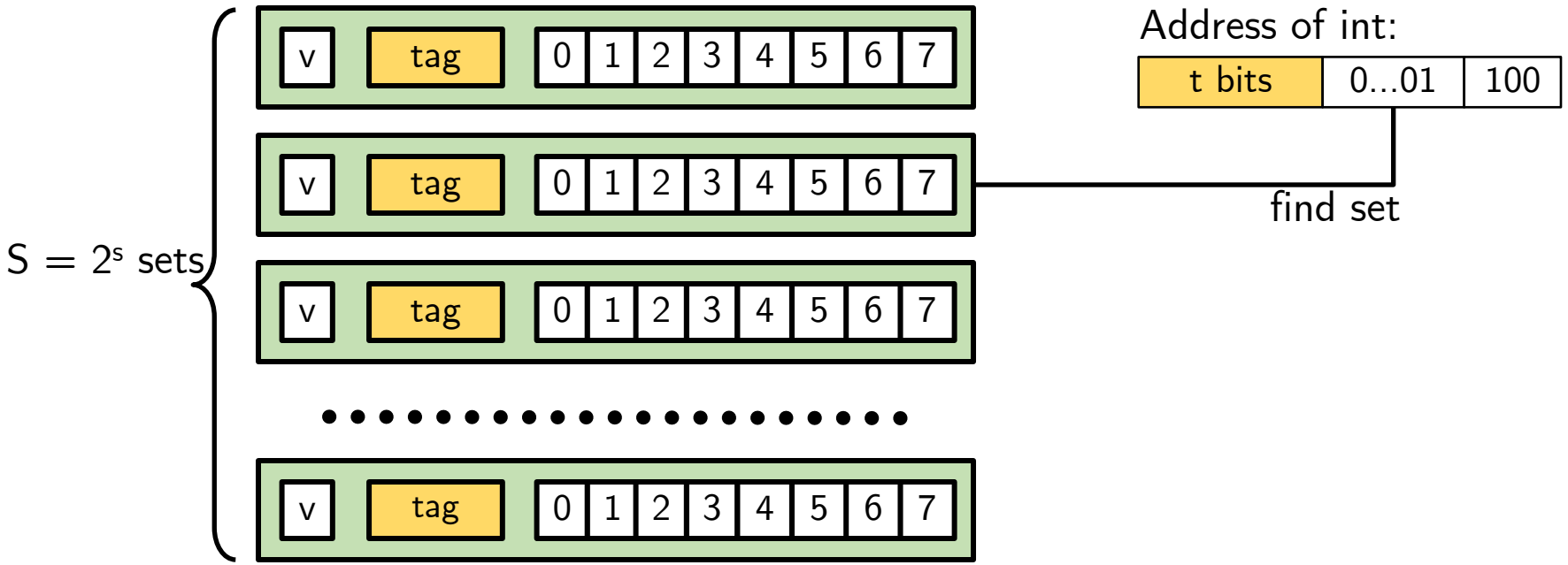


data begins at this offset

***B* = 2^*b* bytes per cache block (the data)**

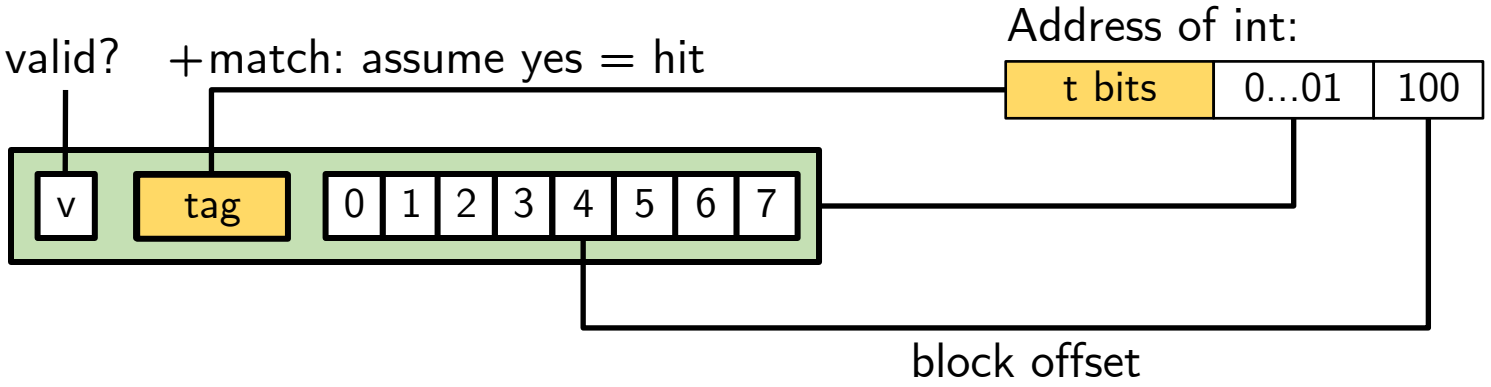
Example: Direct Mapped Cache (E=1)

Direct mapped: One line per set
Assume: cache block size 8 bytes



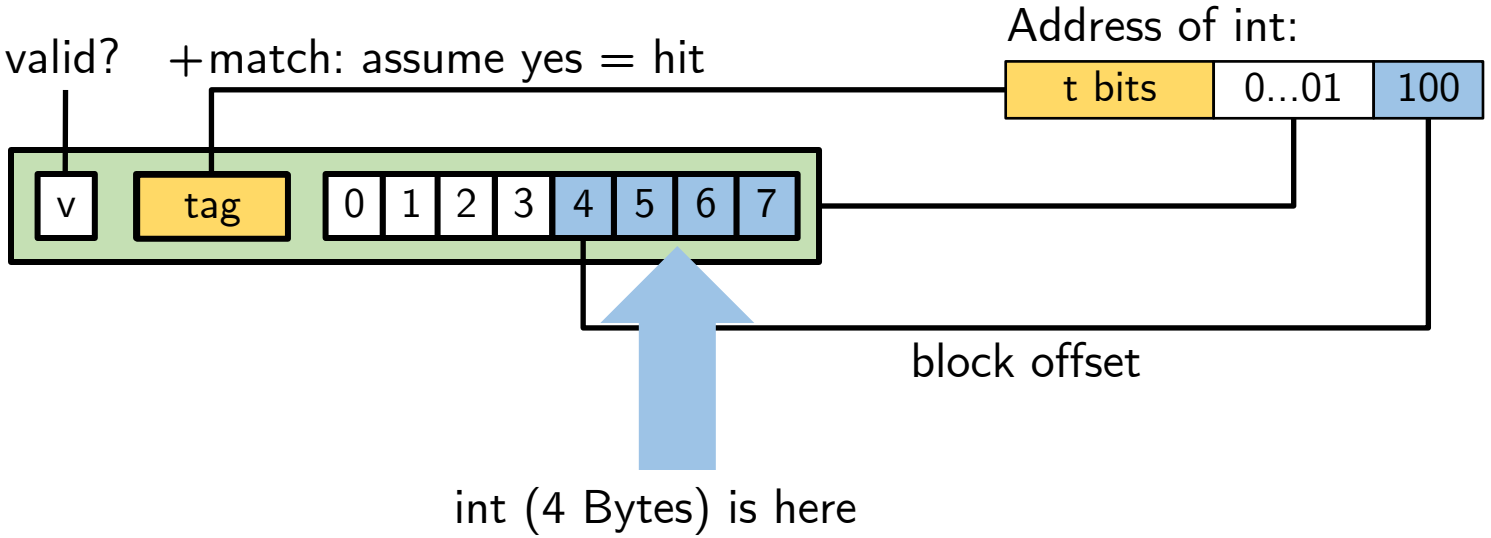
Example: Direct Mapped Cache (E=1)

Direct mapped: One line per set
Assume: cache block size 8 bytes



Example: Direct Mapped Cache (E=1)

Direct mapped: One line per set
Assume: cache block size 8 bytes



If tag doesn't match: old line is evicted and replaced

Direct Mapped Cache Simulation

M = 16 bytes (4-bit addresses),
B = 2 bytes/block,
S = 4 sets,
E = 1 line/set

Memory array: M[16]

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Address trace (reads, one byte per read):

0	$[0\underline{00}0_2]$,	miss
1	$[0\underline{00}1_2]$,	hit: same block w. address 0!
7	$[0\underline{11}1_2]$,	miss: evict address 0!
8	$[1\underline{00}0_2]$,	miss
0	$[0\underline{00}0_2]$	miss: evict address 7!

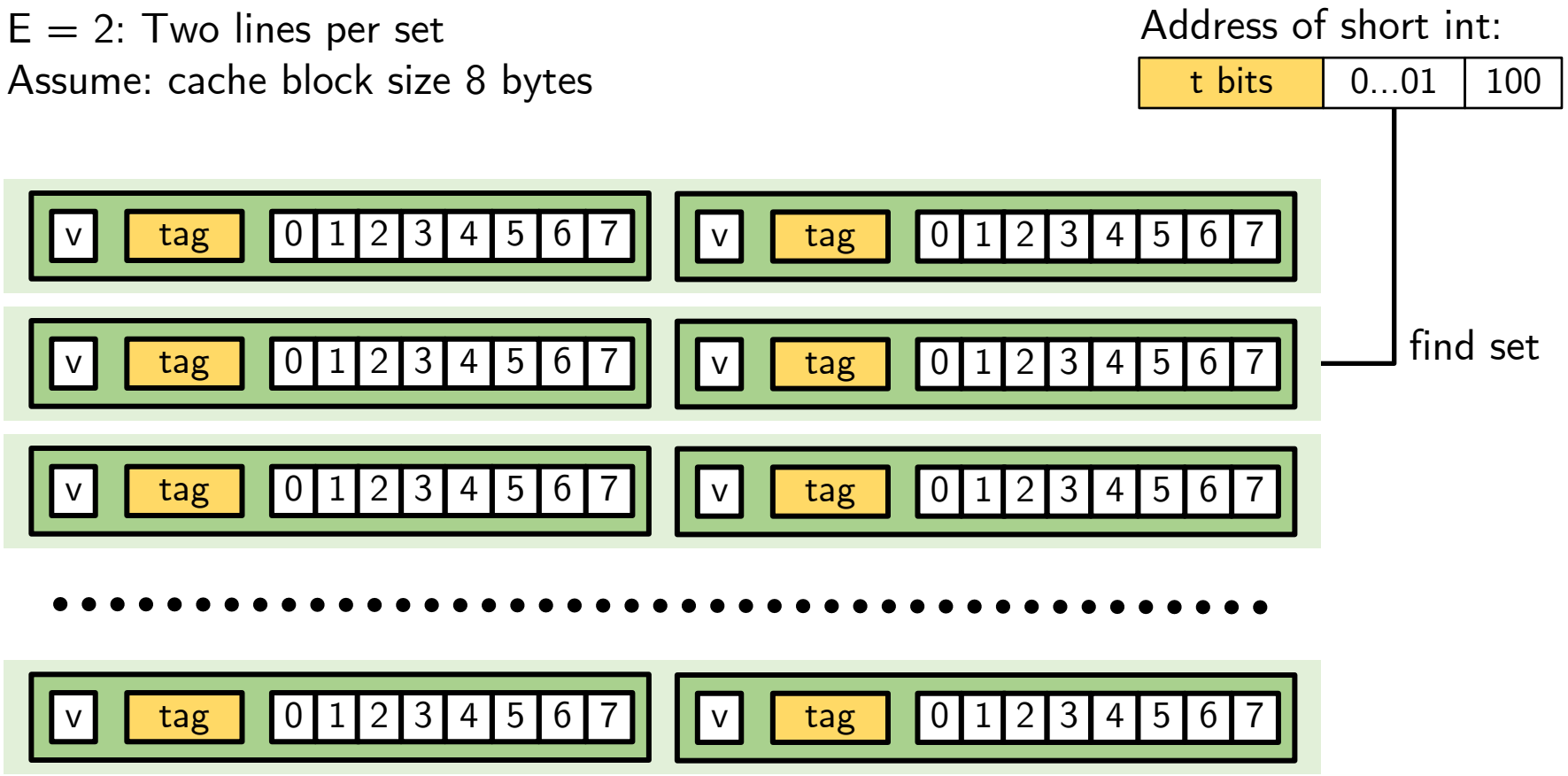
	v	Tag	Block
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3	1	0	M[6-7]

t=1	s=2	b=1
x	xx	x
tag	**set index**	**block offset**

E-Way Set Associative Cache (E=2)

E = 2: Two lines per set

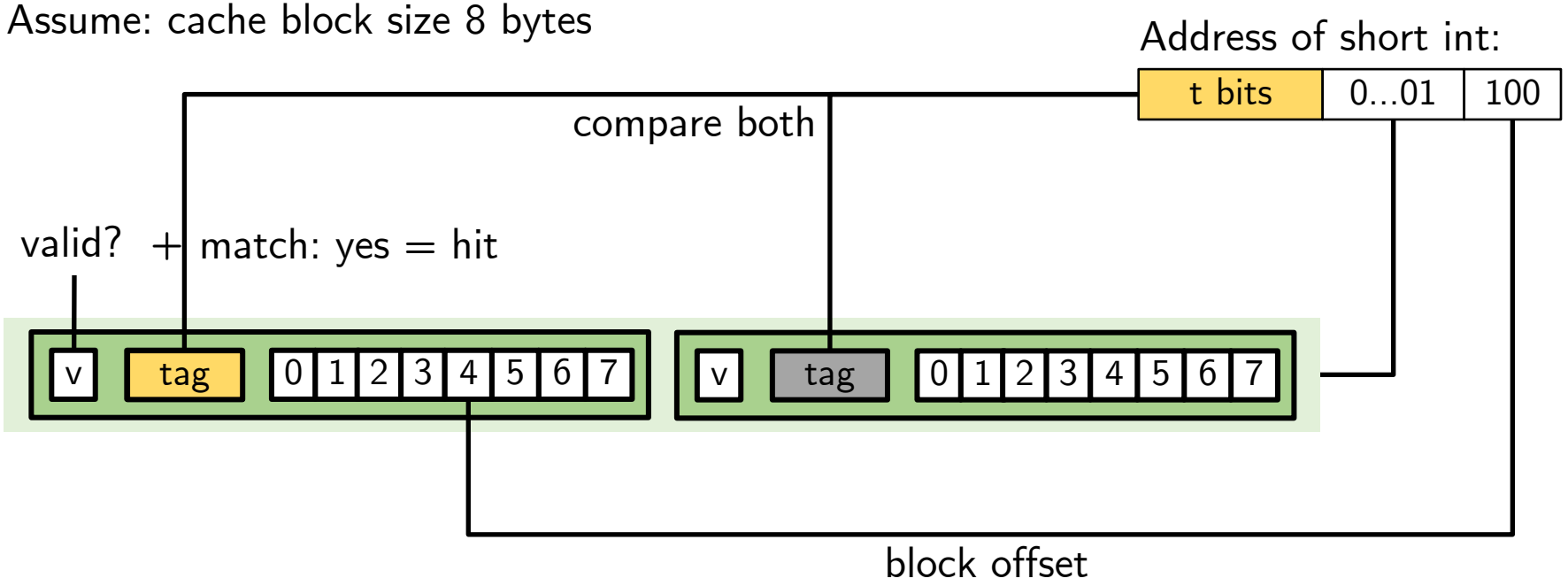
Assume: cache block size 8 bytes



E-Way Set Associative Cache (E=2)

E = 2: Two lines per set

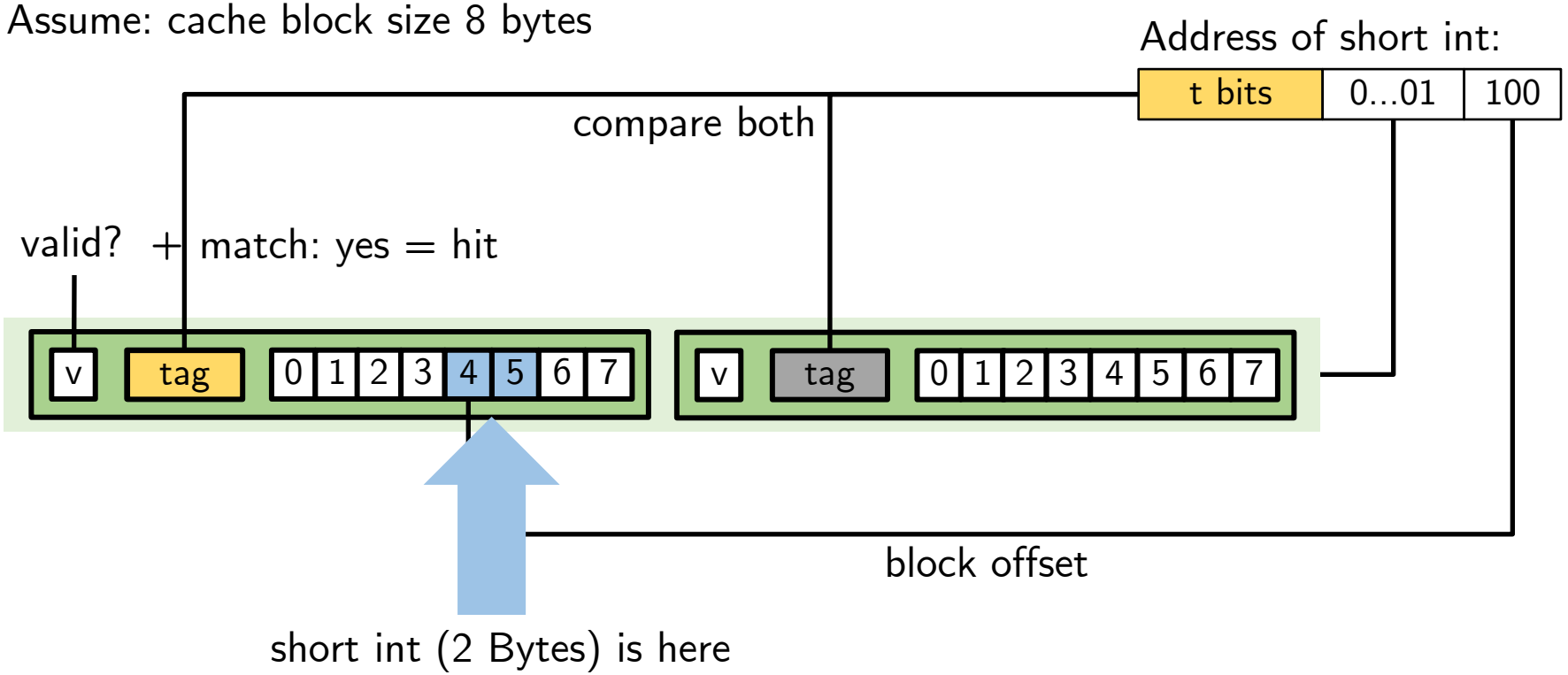
Assume: cache block size 8 bytes



E-Way Set Associative Cache (E=2)

E = 2: Two lines per set

Assume: cache block size 8 bytes



No match: (i.e. cache miss!)

- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), ...

2-Way Set Associative Cache Simulation

M=16 byte (4-bit addresses),
B=2 bytes/block,
S=2 sets,
E=2 lines/set

Memory array: M[16]

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Address trace (reads, one byte per read):

0	$[00\underline{0}0_2]$,	miss
1	$[00\underline{0}1_2]$,	hit
7	$[01\underline{1}1_2]$,	miss
8	$[10\underline{0}0_2]$,	miss
0	$[00\underline{0}0_2]$	hit

	v	Tag	Block
Set 0	1	00	M[0-1]
	1	10	M[8-9]
Set 1	1	01	M[6-7]
	0		

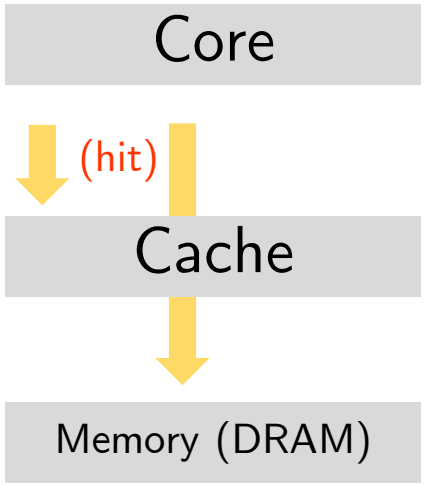
t=2	s=1	b=1
xx	x	x

What about writes?

- Multiple copies of data exist:
  - L1, L2, L3, Main Memory
- What to do on a write-hit?
  - Write-through (write immediately updates the lower level)
  - Write-back (defer write to lower level until replacement of line)
    - Need a dirty bit (line different from lower level or not)
- What to do on a write-miss?
  - No-write-allocate (writes straight to lower level, does not load into cache)
  - Write-allocate (load into cache, update line in cache)
    - Good if more writes to the location follow
- Typical for CPUs
  - **Write-back + Write-allocate**
  - **(don't send data to the next level if you don't have to)**

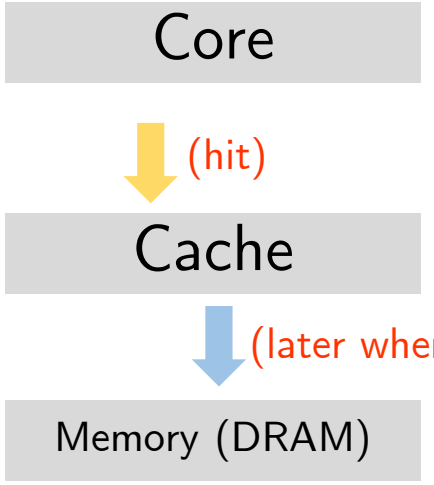
What to do on write-hit, write miss?

What to do on a write-hit?



Write to all levels, in spite of hit.

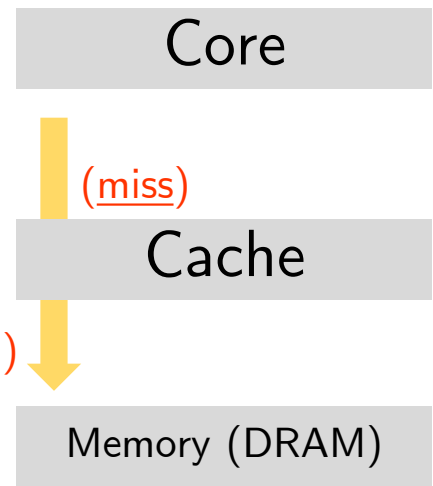
Write Through



Write to cache, evict later

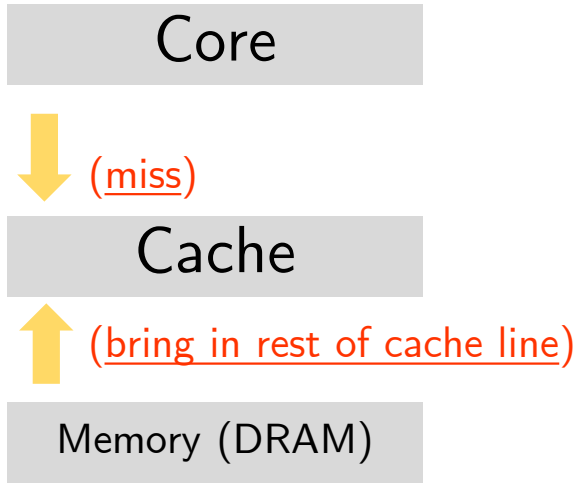
Write-Back

What to do on a write-miss?



Just write to memory, skip cache

Write-No-Allocate



Bring in rest of cache line, write to cache

Write-Allocate

Types of Cache Misses

- Cold (compulsory) miss
  - Cold misses occur because the cache is empty.
- Conflict miss
  - Item fits would have fit in the cache, but was evicted due to restrictions on where it can be placed (due to associativity)
- Capacity miss
  - Occurs when the set of active cache blocks (working set) is larger than the cache.

- E.g. direct mapped cache, address stream: 0, 8, 0, 8, 0, 8
  - First access compulsory miss.
  - If address 0 and 8 have conflicts on the same set, then followed by conflict miss.

- 3C misses.
  - There is another C (Coherence Miss) – not covered in this course.

How to Classify Misses?

- Rerun the trace with an ideal fully-associative cache with infinite capacity.
  - If it still misses in this ideal cache → Cold miss (compulsory).
  - If not, go to the next step.
- Rerun the trace with an ideal fully-associative cache with the same capacity.
  - Same capacity as the original real cache – limit the capacity but relax the associativity.
  - If it still misses in this cache → Capacity miss (not helped even with full associativity).
  - If it now hits → Conflict miss.

- The category basically tells what is the fundamental reason behind this miss.

Direct Mapped Cache Simulation (Again)

M = 16 bytes (4-bit addresses),
B = 2 bytes/block,
S = 4 sets,
E = 1 line/set

Memory array: M[16]

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Address trace (reads, one byte per read):

0	$[0\underline{00}0_2]$,	miss	cold
1	$[0\underline{00}1_2]$,	hit	
7	$[0\underline{11}1_2]$,	miss	cold
8	$[1\underline{00}0_2]$,	miss	cold
0	$[0\underline{00}0_2]$	miss	conflict

	v	Tag	Block
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3	1	0	M[6-7]

t=1	s=2	b=1
x	xx	x
tag	**set index**	**block offset**

Examples of Caching in Memory Hierarchy

Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-8 bytes words	CPU core	1	Compiler
TLB	Address translations	On-Chip TLB	1	Hardware MMU
L1 cache	64-byte blocks	On-Chip L1	4	Hardware
L2 cache	64-byte blocks	On-Chip L2	10	Hardware
Virtual Memory	4-KB pages	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Disk firmware
Network buffer cache	Parts of files	Local disk	10,000,000	NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

Cache Policy

Handling Cache Read

- Instruction Cache and Data Cache
  - Read hit: what we want!
  - Read miss: stall the pipeline, fetch the block from the next level in the memory hierarchy, install it in the cache and send the requested word to the processor, then let the pipeline resume

Handling Cache Write Hits

- Only Data Cache
- Case 1: Write-Through
  - Cache and memory to be consistent
  - always write the data into both the cache block and the next level in the memory hierarchy
  - Speed-up: use write buffer and stall only when buffer is full
- Case 2: Write-Back
  - Write the data only into the cache block
  - Write to memory hierarchy when that cache block is "evicted"
  - Need a dirty bit for each data cache block

Handling Cache Write Misses

- Case 1: Write-Through caches with a write buffer
  - No-write allocate1
  - skip cache write (but must invalidate that cache block since it now holds stale data)
  - just write the word to the write buffer (and eventually to the next memory level)
  - Good for data persistence and immediate data visibility to other units in the system
- Case 2: Write-Back caches
  - Write allocate2
  - Just write the word into the cache updating both the tag and data
  - no need to stall

Replacement Policy

- Direct Mapping
  - Position of each block fixed
  - Whenever replacement is needed (i.e. cache miss → new block to load), the choice is obvious and thus no "replacement algorithm" is needed
- Associative and Set Associative
  - Need to decide which block to replace
  - Keep/retain ones likely to be used in near future again
- Strategy 1: Least Recently Used (LRU)
  - e.g. for a 4-block/set cache, use a $\log_2 4 = 2$ bit counter for each block
  - Reset the counter to 0 whenever the block is accessed, counters of other blocks in the same set should be incremented
  - On cache miss, replace/ uncache a block with counter reaching 3
- Strategy 2: Random Replacement
  - Choose random block
  - Easier to implement at high speed

Optimize Matrix Computation for Cache

Cache Performance Metrics

- Memory stall cycles:

$$memory\ stall\ cycles = \frac{memory\ access}{program} \times Miss\ Rate \times miss\ penality$$

- Miss Rate: Fraction of accesses not found in cache = 1 – hit rate
  - 3-10% for L1
  - Can be quite small (e.g., < 1%) for L2, depending on size, etc.
- Impact of miss rate is determined by "hit time" and "miss penalty"
- Hit Time: Time to deliver a line in the cache to the processor
  - Includes time to determine whether the line is in the cache
  - 1-4 clock cycle for L1
  - 10-20 clock cycles for L2
- Miss Penalty: Additional time required because of a miss
  - Typically, 50-200 cycles for main memory

Practice

Assume the miss rate of an instruction cache is 2% and the miss rate of the data cache is 4%. If a processor has a CPI of 2 without any memory stalls, and the miss penalty is 100 cycles for all misses, compute the CPI of this computer, and determine how much faster a processor would run with a perfect cache that never missed. Assume the frequency of all loads and stores is 36%.

- Miss cycles from instruction cache: $2\% \times 100 = 2$
- Miss cycles from data cache: $36\% \times 4\% \times 100 = 1.44$
- Total CPI: $2 + 2 + 1.44 = 5.44$

- Speedup from a perfect cache: $5.44 \div 2 = 2.72 \times$

Let's think about those numbers

- Huge difference between a hit and a miss
  - Could be 100x, if miss all cache levels

- Would you believe 99% hits is twice as good as 97%?
  - If the miss penalty is high enough, this is true!
  - Consider: cache hit time of 1 cycle, miss time of 100 cycles.

  - Average access time:

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

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

- **This is why "miss rate" is often preferred over "hit rate".**

How Important are Misses?

- Additional Metric: Misses per thousand instruction.
  - How many misses / count of dynamically executed instructions * 1000
- Capture how important misses are, in addition to how poorly the cache is.
- Example:
  - Miss rate of L2: 20% (very high)
  - Misses per thousand instructions: 1.0
- Should we be worried about the miss rate affecting our performance?
  - No

Writing Cache Friendly Code

- Make the common case go fast
  - Focus on the inner loops of the core functions

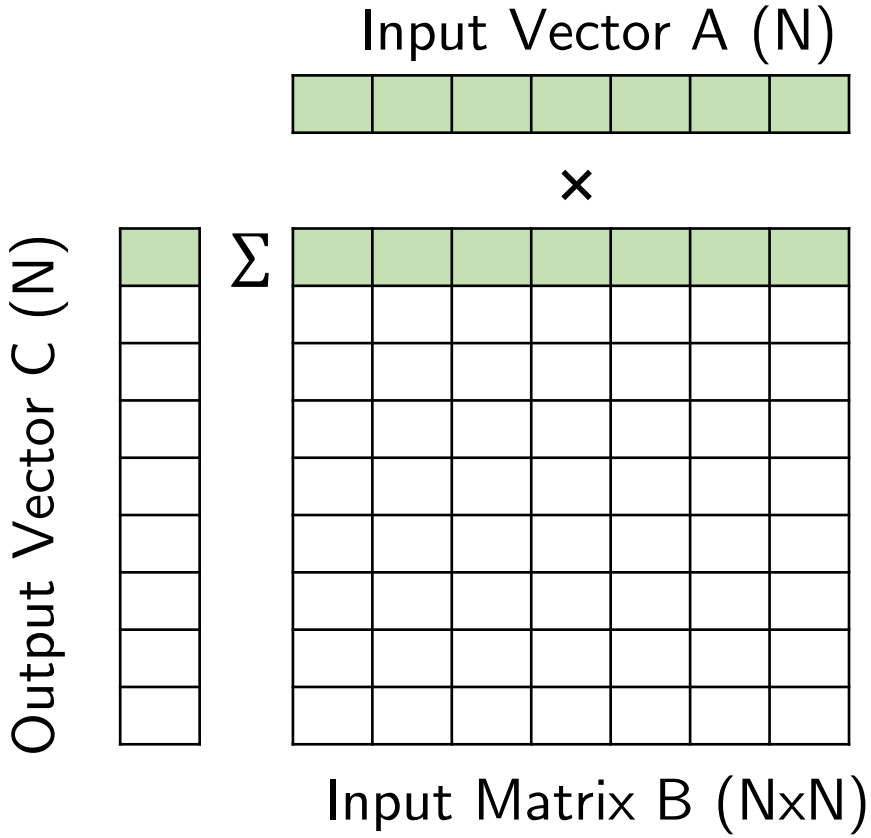
- Minimize the misses in the inner loops
  - Repeated references to variables are good (temporal locality)
  - Stride-1 reference patterns are good (spatial locality)

- Try to reuse Data as much as possible!
  - E.g., through "tiling" (book calls this "Blocking" but that's super confusing)

Matrix-Vector Multiplication Example

```
int A[N];
int B[N][N];
int C[N];

for (j = 0; j < N; j++) {
  sum=0;
  for (i = 0; i < N; i++) {
    sum += A[i] * B[j][i];
  }
  C[j] = sum;
}
```



Assume N×sizeof(int) >> cache size, 64-byte lines, ... what is miss rate of each access?

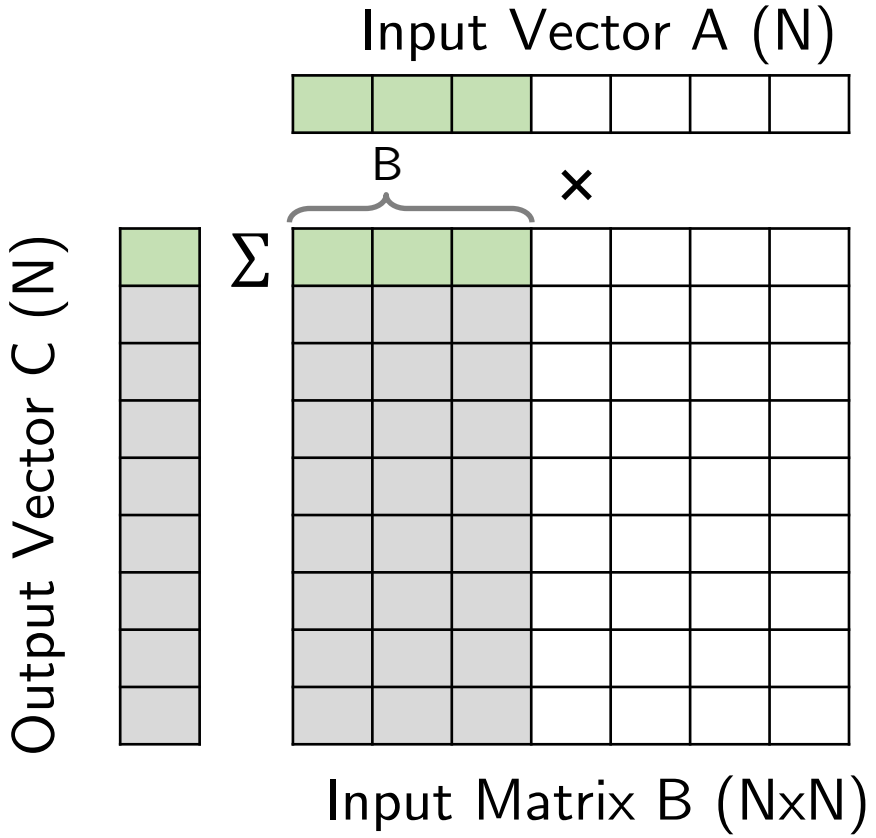
How can we do better than this?

Tiling: Transformation for Locality

```
int A[N];
int B[N][N];
int C[N];

#define T ... //should divide N evenly

for(i_tile=0; i_tile<N; i_tile+=T){
  for (j = 0; j < N; j++) {
    sum=C[j]; //get partial sum
    for (i=i_tile; i<i_tile+T;i++){
      sum += A[i] * B[j][i];
    }
    C[j] = sum;
  }
}
```

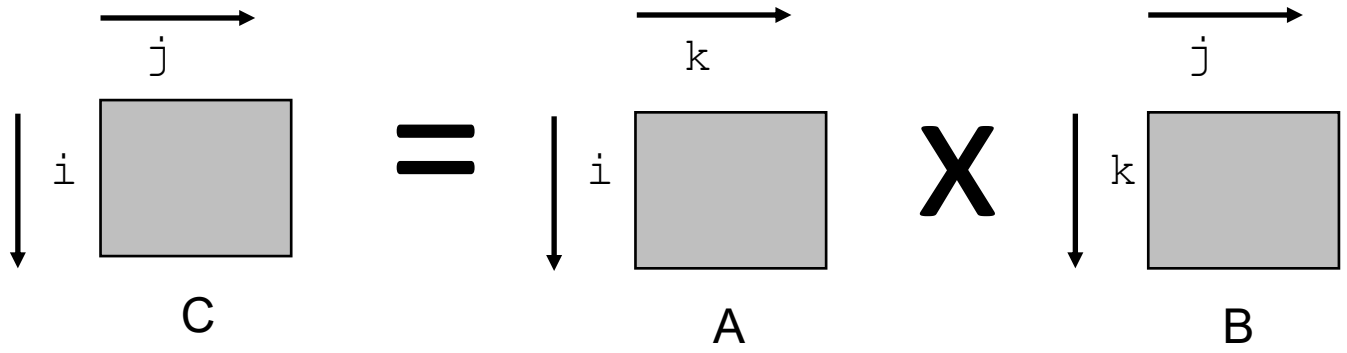


Assume N*sizeof(int) >> cache size, 64-byte lines, ... what is miss rate of each access?

Matrix Multiplication Example

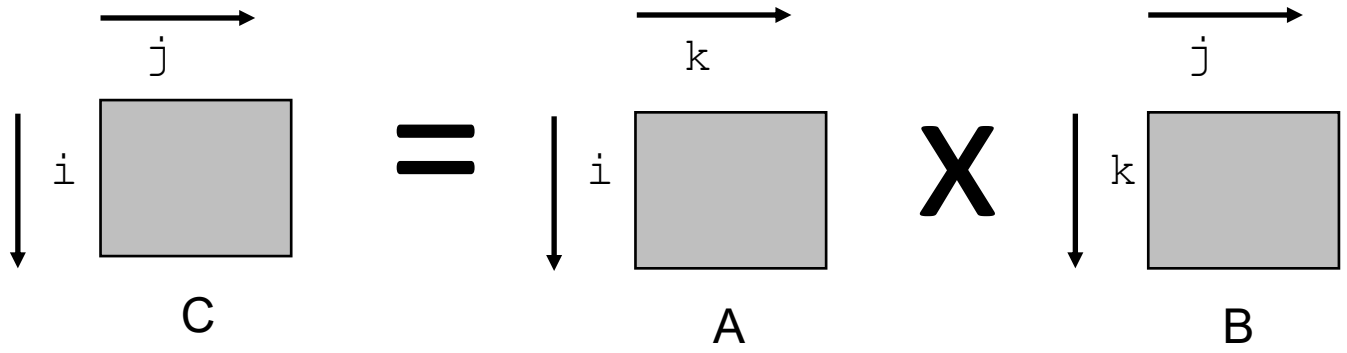
- Description:
  - Multiply N x N matrices
  - Matrix elements are doubles (8 bytes)
  - $O(N^3)$ total operations
  - N reads per source element
  - N values summed per destination
    - but may be able to hold in register

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    for (k=0; k<n; k++)
      c[i][j] += a[i][k] * b[k][j];
```



Miss Rate Analysis for Matrix Multiply

- Assume:
  - **Cache Block (cache line) size = 64B (big enough for 8 doubles)**
  - Matrix dimension (N) is very large
    - Approximate 1/N as 0.0
  - **Cache is not even big enough to hold multiple rows**
- Analysis Method:
  - Look at access pattern of inner loop



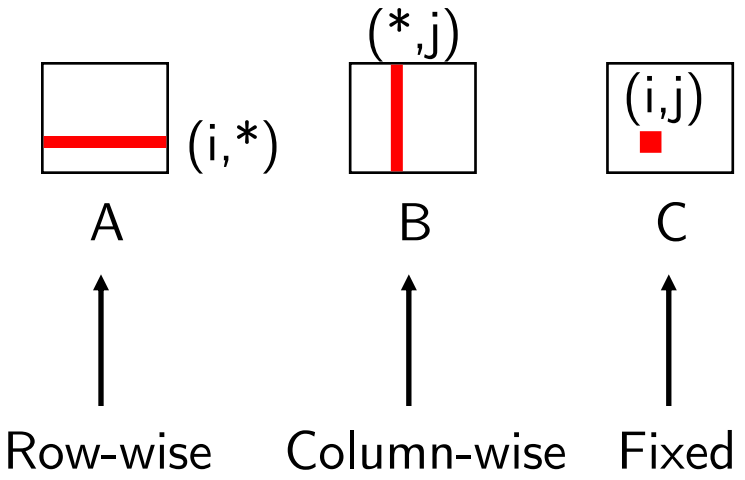
Layout of C Arrays in Memory (review)

- **C arrays allocated in row-major order**
  - each row in contiguous memory locations
- **Stepping through columns in one row:**
  - `for (i = 0; i < N; i++)`
    `sum += a[0][i];`
  - accesses successive elements
  - if block size (B) > sizeof(a_{ij}) bytes, exploit spatial locality
    - miss rate = sizeof(a_{ij}) / B
- **Stepping through rows in one column:**
  - `for (i = 0; i < n; i++)`
    `sum += a[i][0];`
  - accesses distant elements
  - no spatial locality!
    - miss rate = 1 (i.e. 100%)

Matrix Multiplication (ijk)

```
/* 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;
  }
}
```

Inner loop:

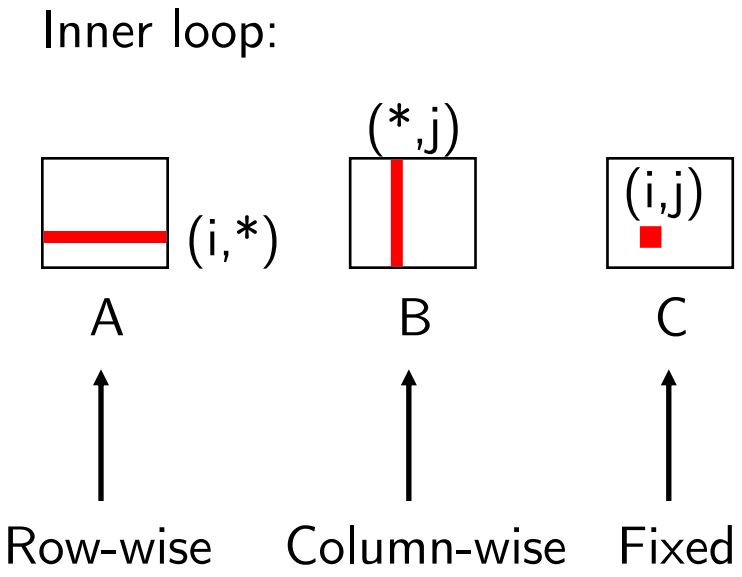


Misses per inner loop iteration:

A	B	C
0.125	1.0	0.0

Matrix Multiplication (jik)

```
/* jik */
for (j=0; j<n; i++)  {
  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;
  }
}
```



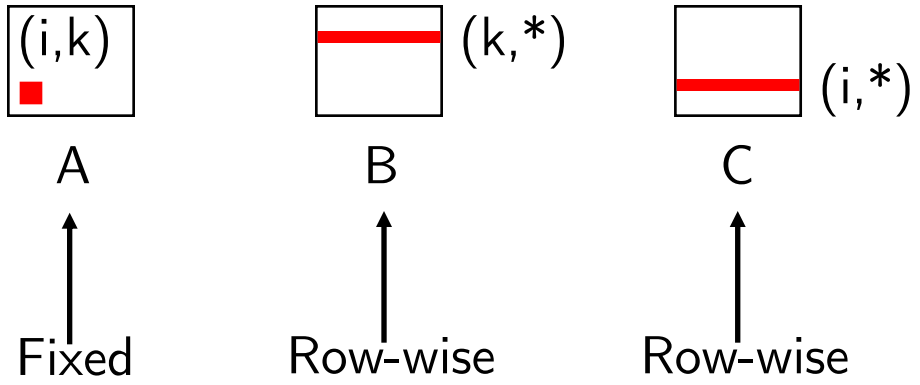
Misses per inner loop iteration:

A	B	C
0.125	1.0	0.0

Matrix Multiplication (kji)

```
/* 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];
  }
}
```

Inner loop:



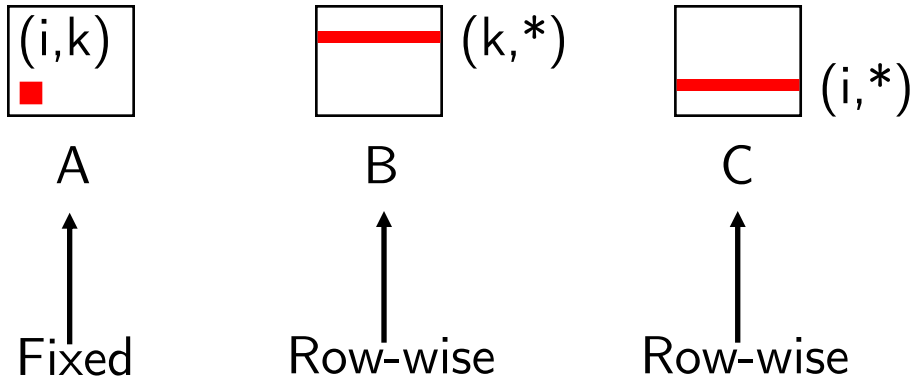
Misses per inner loop iteration:

A	B	C
0.0	0.125	0.125

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];
  }
}
```

Inner loop:



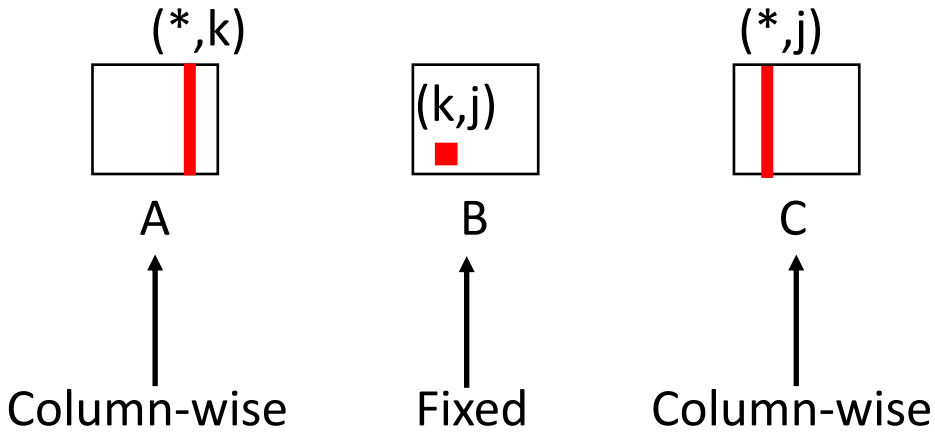
Misses per inner loop iteration:

A	B	C
0.0	0.125	0.125

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;
  }
}
```

Inner loop:



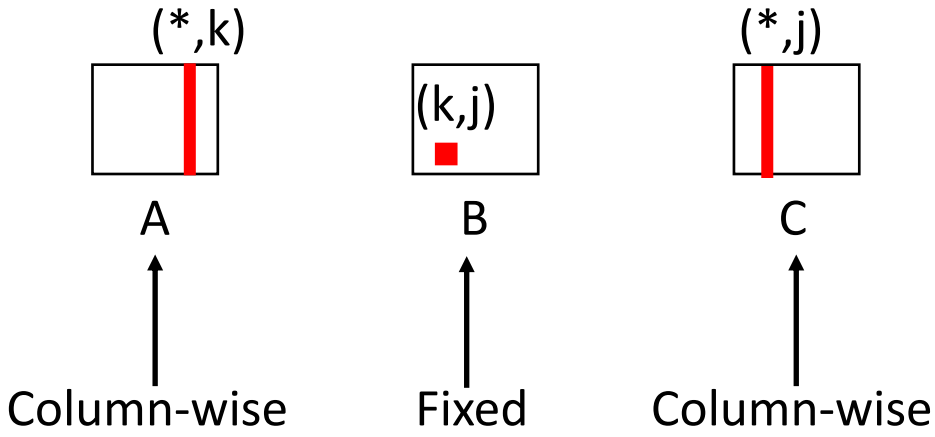
Misses per inner loop iteration:

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;
  }
}
```

Inner loop:



Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Summary of Matrix Multiplication

```
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;
 }
}
```

ijk (& jik):

- 2 loads, 0 stores
- misses/iter = 1.125

```
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];
 }
}
```

kij (& ikj):

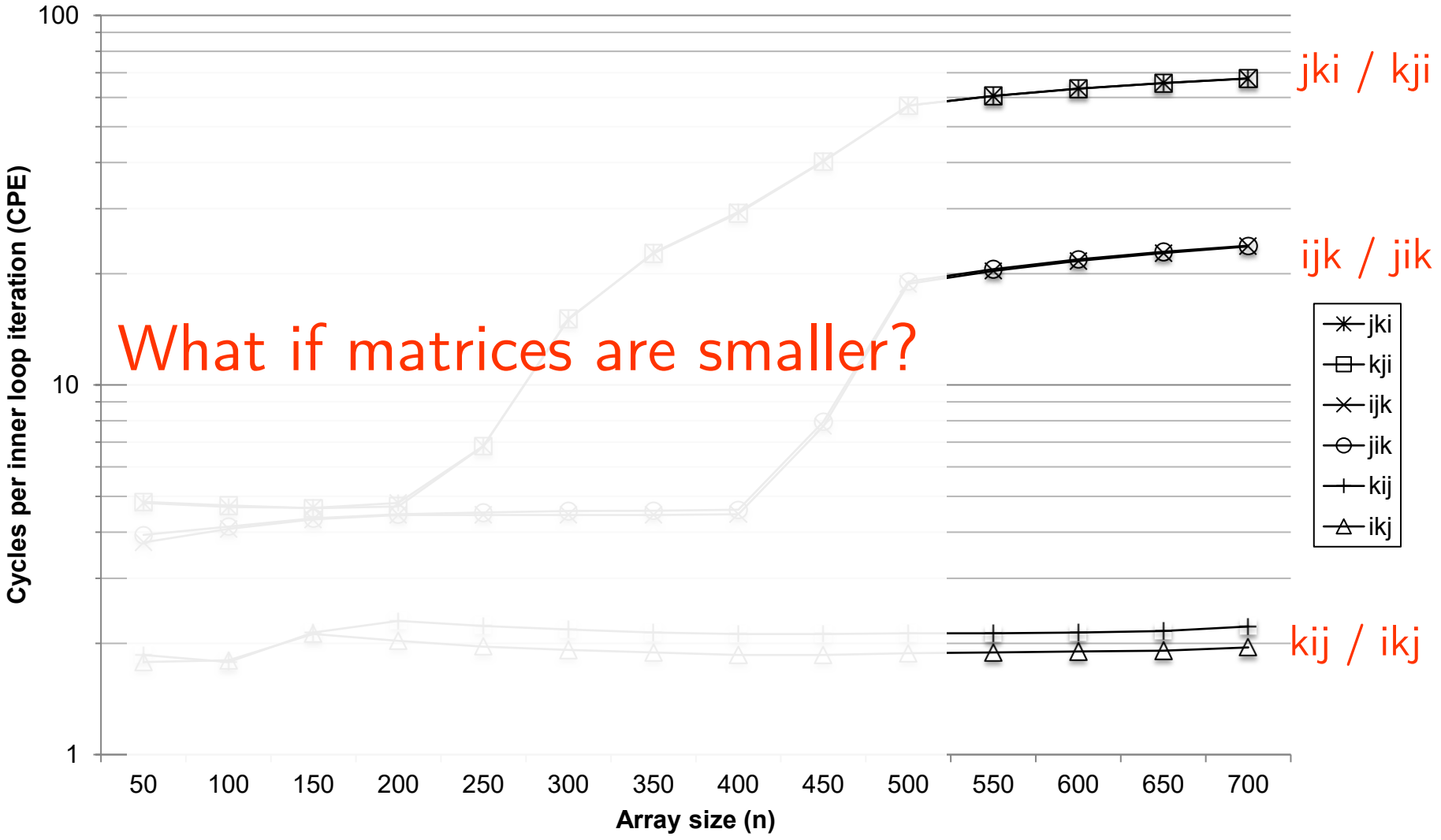
- 2 loads, 1 store
- misses/iter = 0.25

```
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;
 }
}
```

jki (& kji):

- 2 loads, 1 store
- misses/iter = 2.0

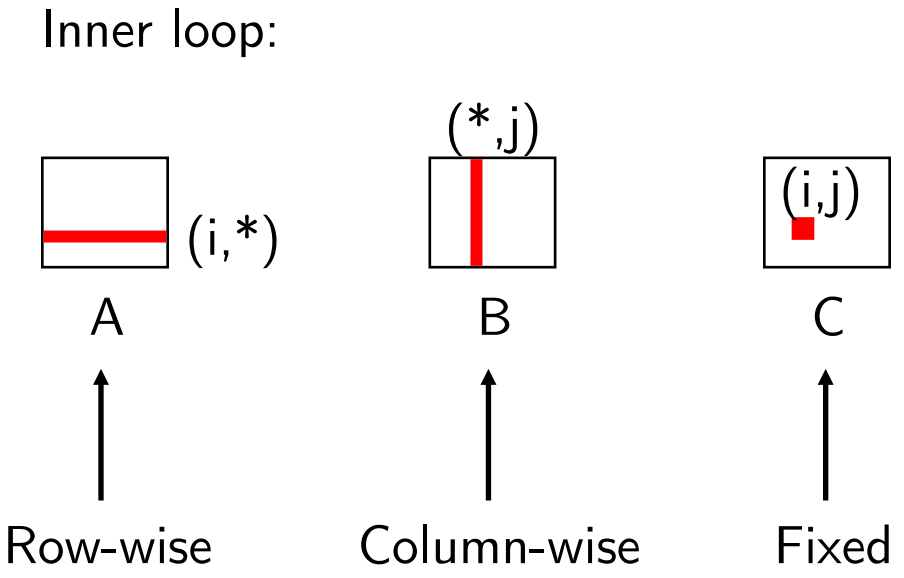
Core i7 Matrix Multiply Performance



Small Matrix Multiplication (ijk)

- Insight: All data is only read once.
- Misses = Total Cache Blocks / Total Elements

```
/* 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 iter (big array):

A	B	C
0.125	1.0	0.0

Total misses/iter: 1.125

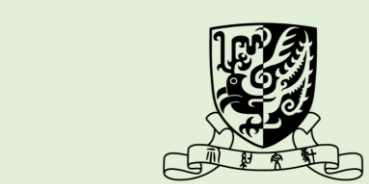
Misses per iter (small array):

Total Blocks: n^2 * 3 arrays / (8 elem/block)

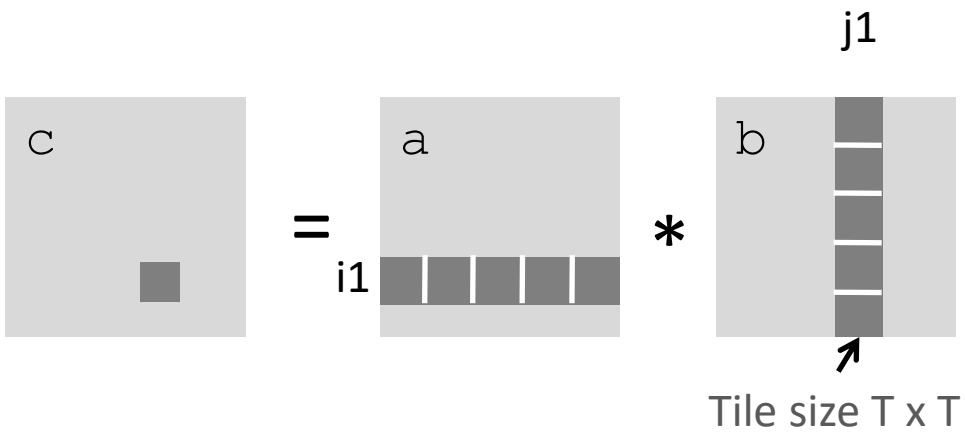
Total Iters: n^3

Total misses/iter: $(n^2*3/8)/n^3 = 3/(8n)$

(e.g. N=48
Total data =55KB
Misses/iter: 0.0078125)



Tiled Matrix Multiplication




```
/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
  for (int i_tile = 0; i_tile < n; i_tile+=T)
    for (int j_tile = 0; j_tile < n; j_tile+=T)
      for (int k_tile = 0; k_tile < n; k_tile+=T)
        /* T x T mini matrix multiplications */
        for (int i = i_tile; i < i_tile+T; i++)
          for (int j = j_tile; j < j_tile+T; j++)
            for (int k = k_tile; k < k_tile+T; k++)
              c[i][j] += a[i][k] * b[k][j];
}
```

Overall misses/iter: ~ 3/(8*T)

Cache Summary

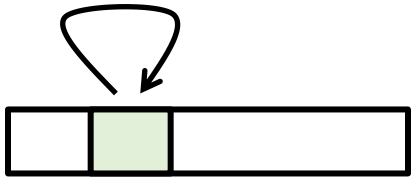
- Cache memories can have significant performance impact

- You can write your programs to exploit this!
  - Focus on the inner loops, where bulk of computations and memory accesses occur.
  - Try to maximize spatial locality by reading data objects with sequentially with stride 1.
  - Try to maximize temporal locality by using a data object as often as possible once it's read from memory.

Locality

- Principle of Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently

- Temporal locality:
  - Recently referenced items are likely to be referenced again in the near future



- Spatial locality:
  - Items with nearby addresses tend to be referenced close together in time