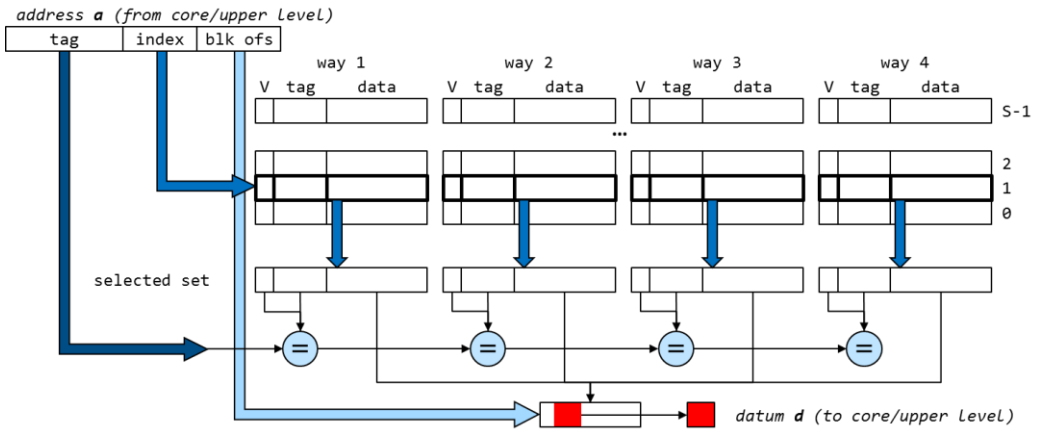
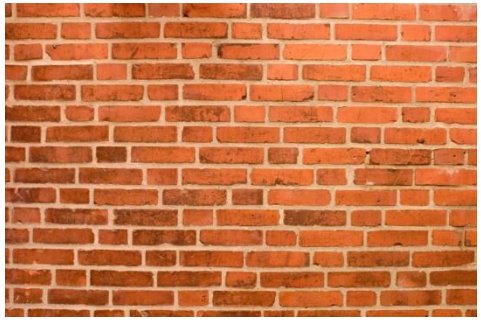
The Storage Hierarchy

Cache Memories



Module Outline

- The Memory Wall
- Locality of Reference
- Caching in the Memory Hierarchy
- Cache Organization
  - Direct-mapped caches
  - Fully-associative caches
  - N-way set associative caches
  - A common framework for memory hierarchies
- Optimizing for the memory hierarchy
- Module Summary

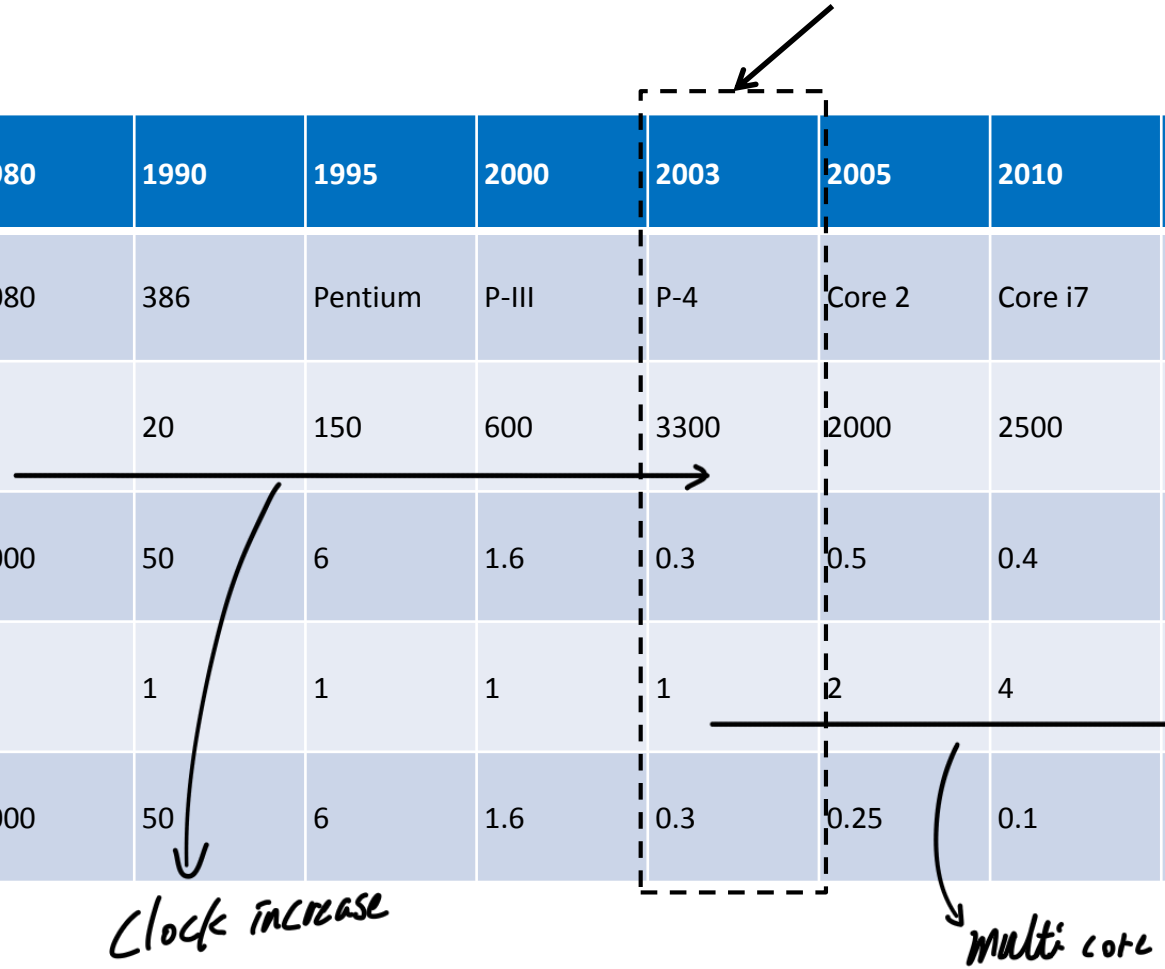


The Memory Wall

CPU Clock Rates

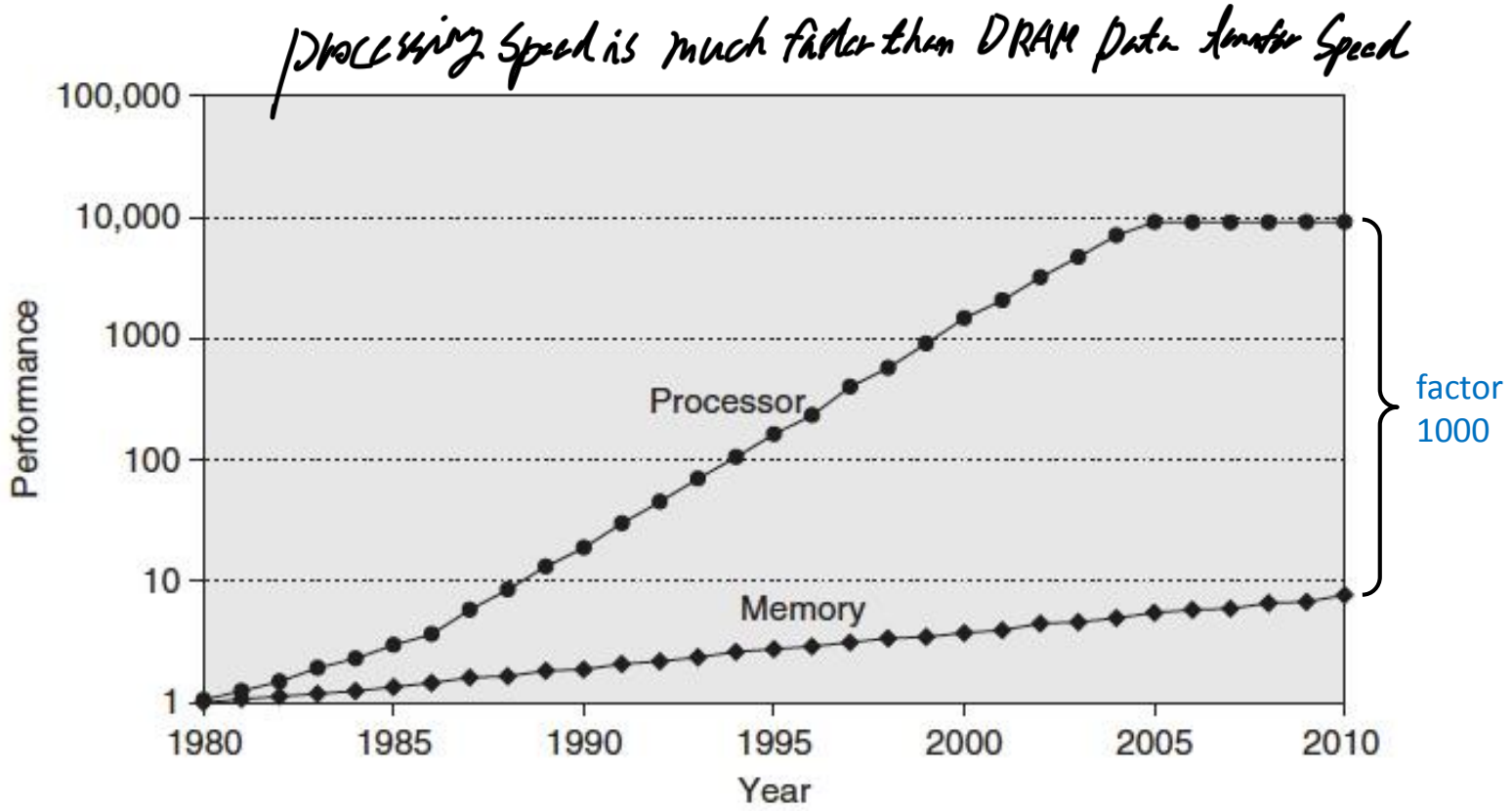
Inflection point in computer history when designers hit the "Power Wall"

	1980	1990	1995	2000	2003	2005	2010	2023	2023:1980
CPU	8080	386	Pentium	P-III	P-4	Core 2	Core i7	Core i9	
Clock (MHz)	1	20	150	600	3300	2000	2500	**3600**/4500	3600/4500
Cycle time (ns)	1000	50	6	1.6	0.3	0.5	0.4	0.3	3,300
Cores	1	1	1	1	1	2	4	8	8
Effective cycle time (ns)	1000	50	6	1.6	0.3	0.25	0.1	0.035	30,000



The Problem: The Memory Wall

- Huge performance gap between modern CPUs and memory
  - single-core performance vs DRAM performance (latency)



Source: Computer Architecture: A Quantitative Approach, Hennessy & Patterson, Elsevier, 2012

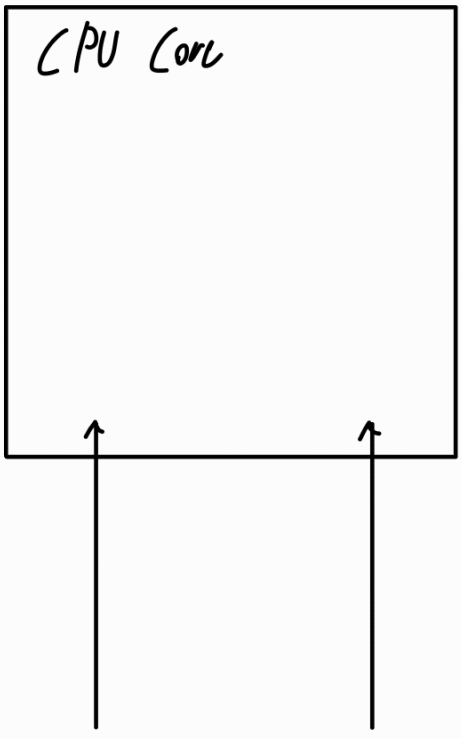
The Problem: The Memory Wall

- Recently, CPU performance has peaked, while memory keeps getting faster

  Can we sit the problem out?

- No: even though single-core CPU performance is stagnating, the memory system has to support more and more cores
  - Intel Core i9-13900K with 24 (8 performance, 16 efficient) cores @ 3.0/2.2GHz (with Turbo Boost up to 5.8/4.3GHz; released Q4 2022)
    - memory references per cycle:
      up to two 64-bit data memory references per cycle and performance core
      up to one 64-bit data memory references per cycle and efficient core:
      8*2*3.0G + 16*1*2.2G = 83.2 billion 64-bit data references per second
    - one 128-bit instruction references per cycle and core:
      8*3.0G + 16*2.2G = 59.2 billion 128-bit instruction references per second
    - = 82.2G*64 + 59.2G*128 = 1.6 TB/sec peak memory bandwidth
    - DDR5 DRAM peak bandwidth: 64GB/sec

    factor 25
  - AMD Threadripper 3990X (64 cores @ 2.9GHz): ~9 TB/sec peak mem bw

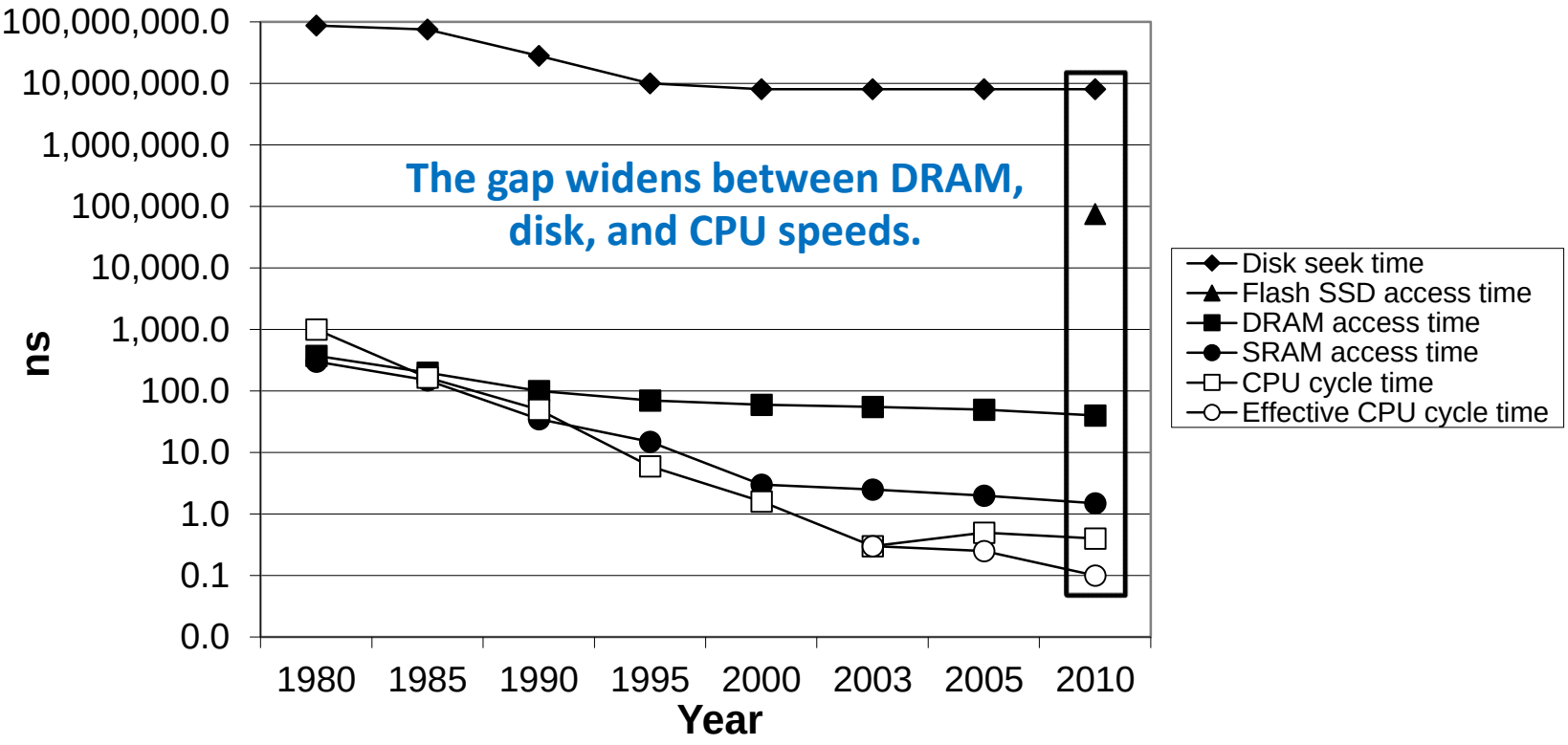


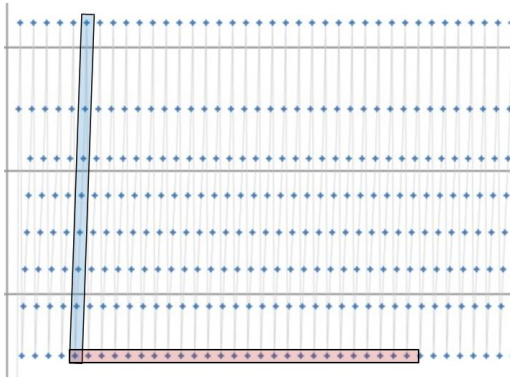
Hyper threading : run double thread
in single core

The Problem: The Memory Wall

- **Basic rules for memory technologies**
  - the bigger the slower
  - the faster the more expensive

Disk – RAM – CPU gap ↑





Locality of Reference

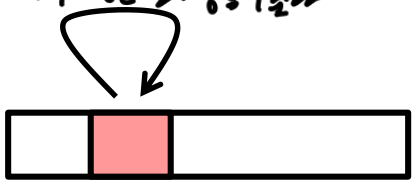
The Principle of Locality

- The key to bridging this CPU-Memory gap is a fundamental property of computer programs known as **locality**

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

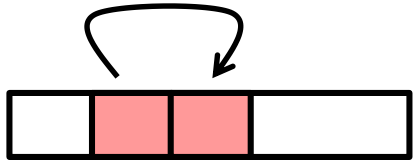
- **Temporal locality (시간 구역성)**: 한번 방문했다면, 빠른 시간내에 다시 방문할 가능성이 높다

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



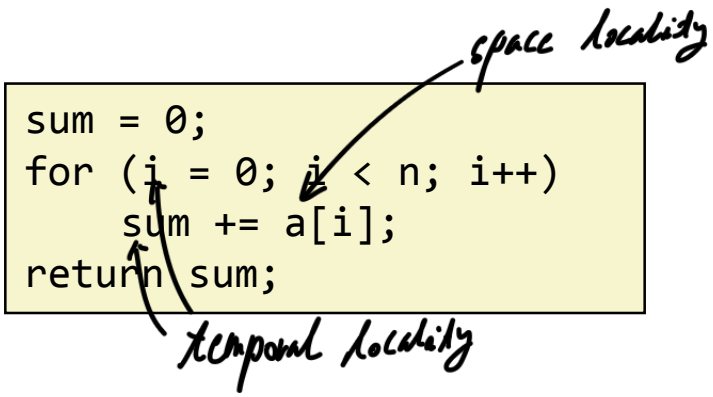
- **Spatial locality (공간 구역성)**: 방문한곳 근처를 방문할 가능성이 높다

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



Locality Example

Mandalorian



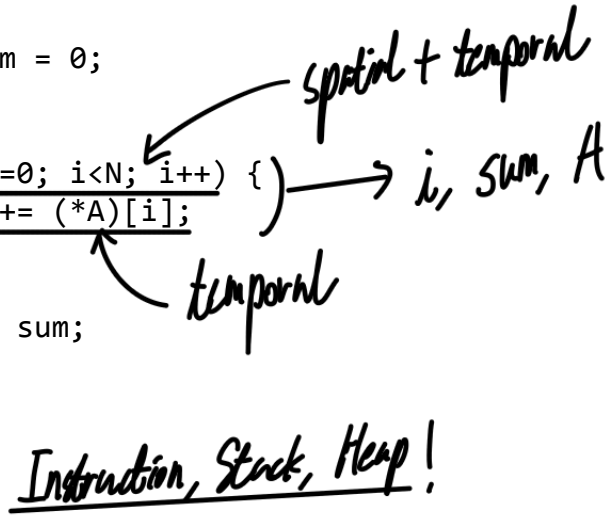
- Data references
  - Reference array elements in succession (stride-1 reference pattern). — Spatial locality
  - Reference variable sum each iteration. — Temporal locality
- Instruction references
  - Reference instructions in sequence. — Spatial locality
  - Cycle through loop repeatedly. — Temporal locality

Examining Locality

- Consider the following three functions

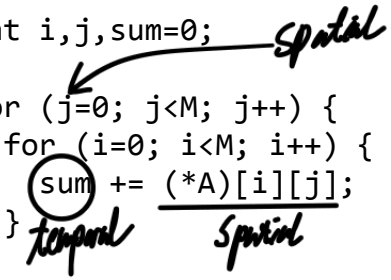
```
#define N 1024
#define M 32

typedef int A1D[N];
typedef int A2D[M][M];
```

```
int array1d_sum(A1D *A)
{
  int sum = 0;
  int i;

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

  return sum;
}
```



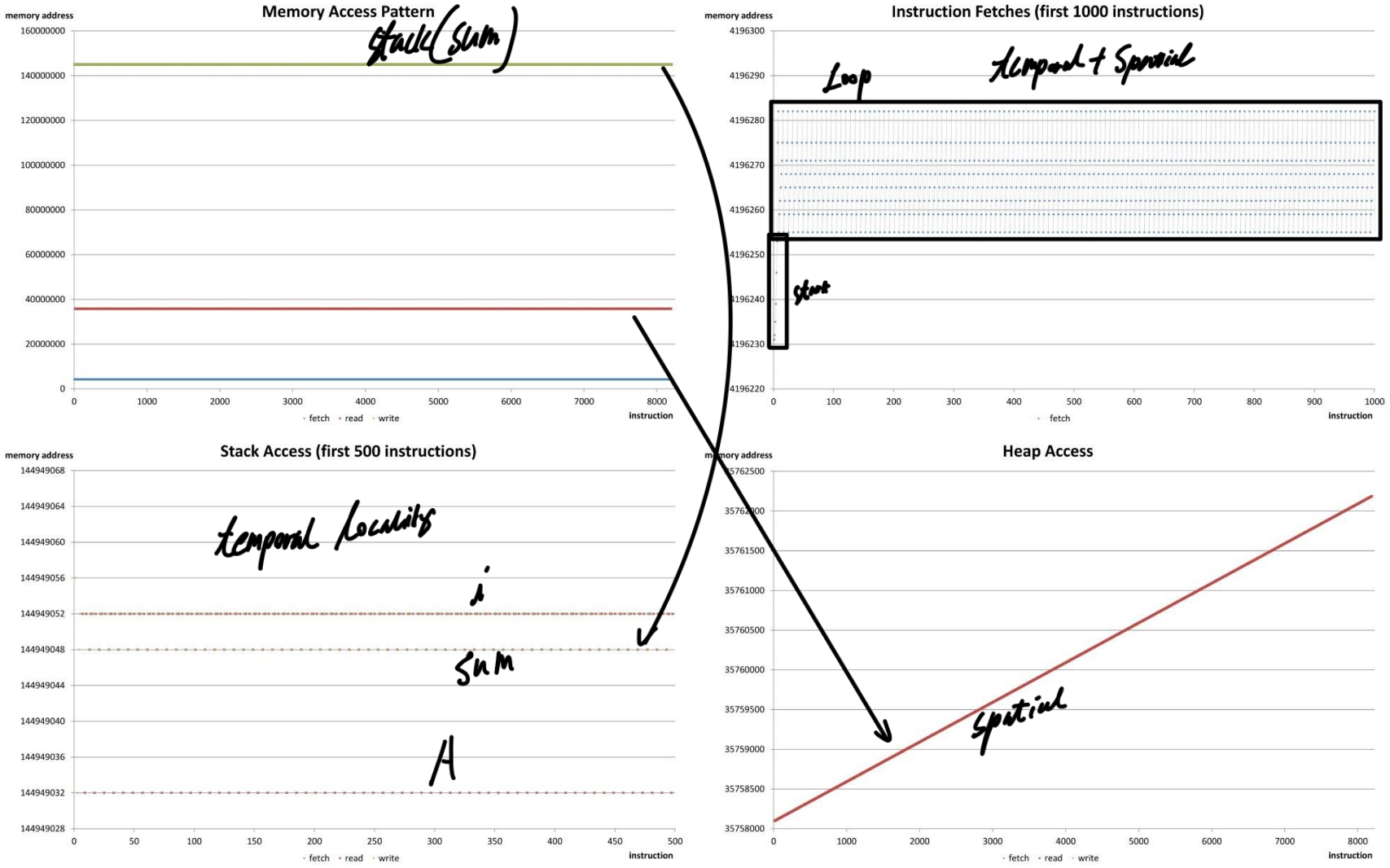

```
int array2d_sum_rm(A2D *A)
{
  int i,j,sum=0;

  for (i=0; i<M; i++) {
    for (j=0; j<M; j++) {
      sum += (*A)[i][j];
    }
  }

  return sum;
}
```

```
int array2d_sum_cm(A2D *A)
{
  int i,j,sum=0;

  for (j=0; j<M; j++) {
    for (i=0; i<M; i++) {
      sum += (*A)[i][j];
    }
  }

  return sum;
}
```



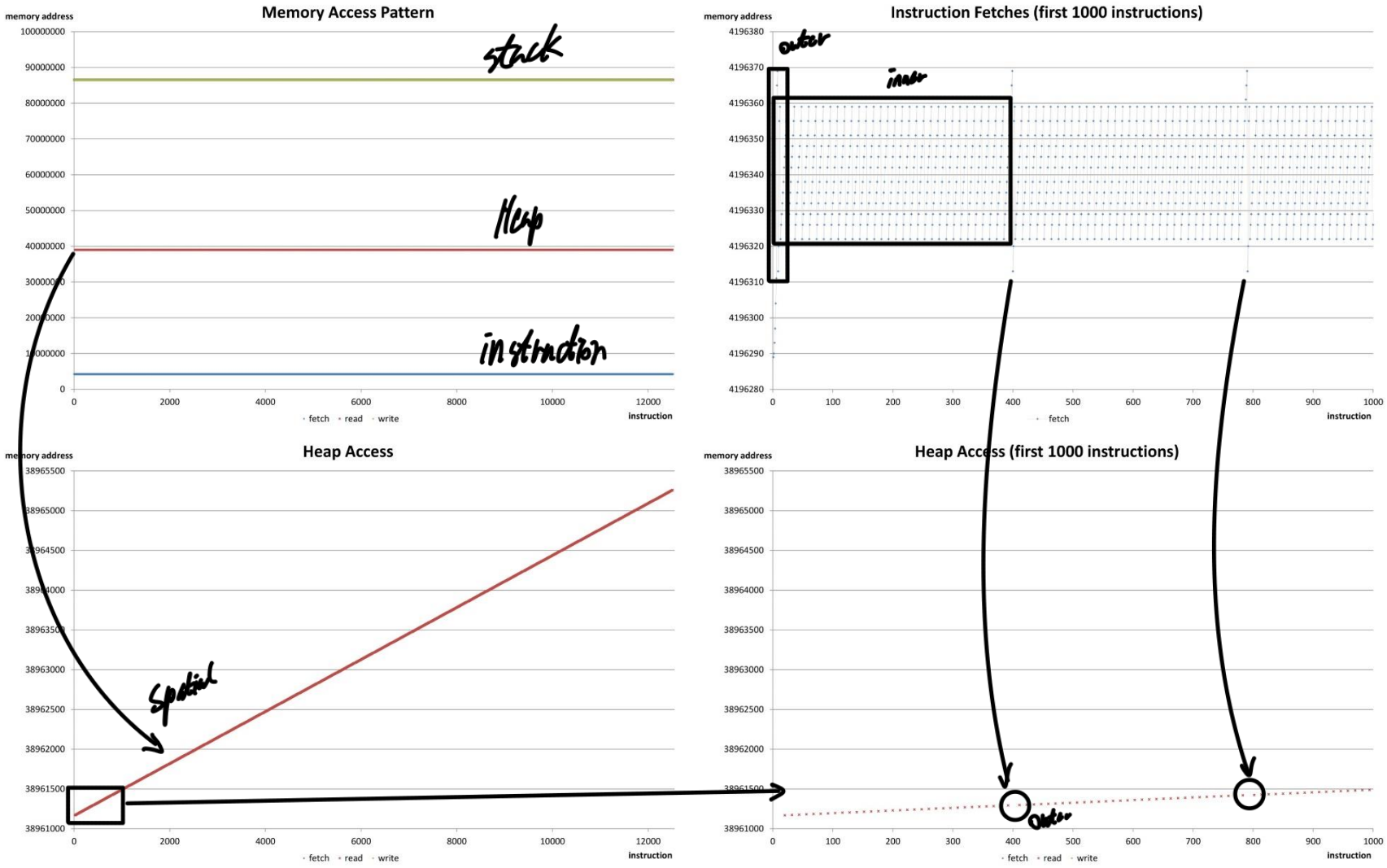
Locality: Memory Access Patterns

- array1d_sum()



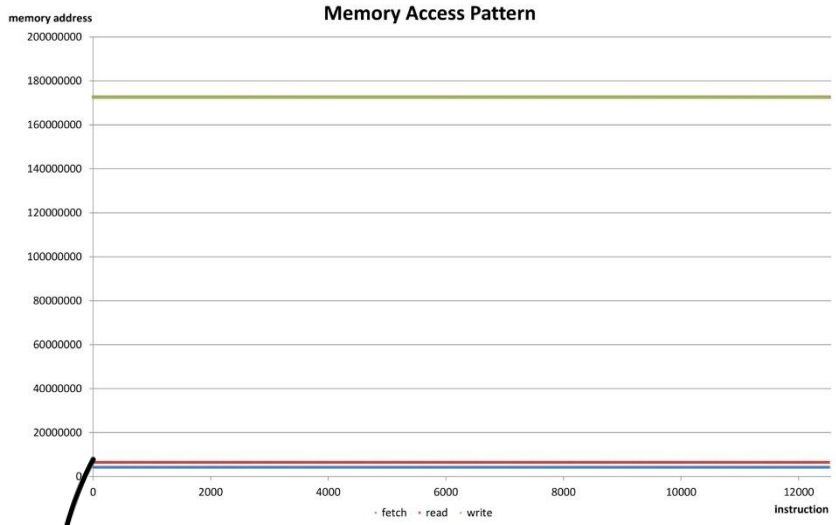
Locality: Memory Access Patterns

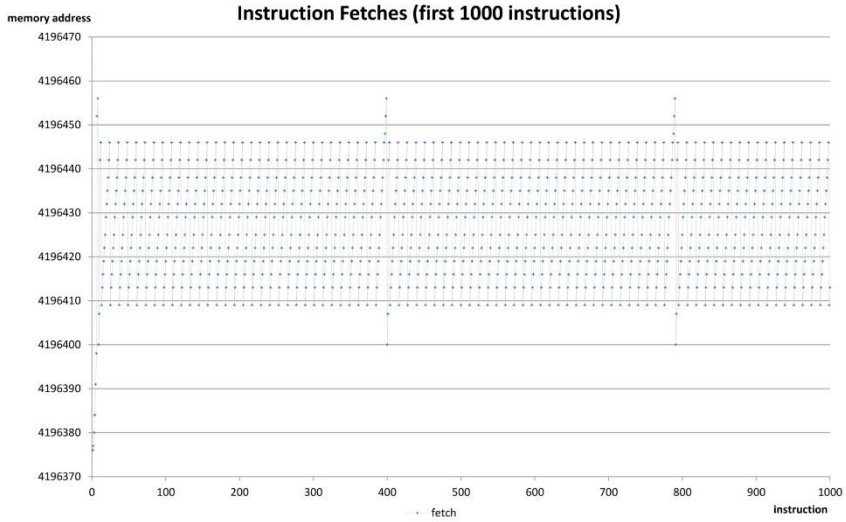
- array2d_sum_rm()



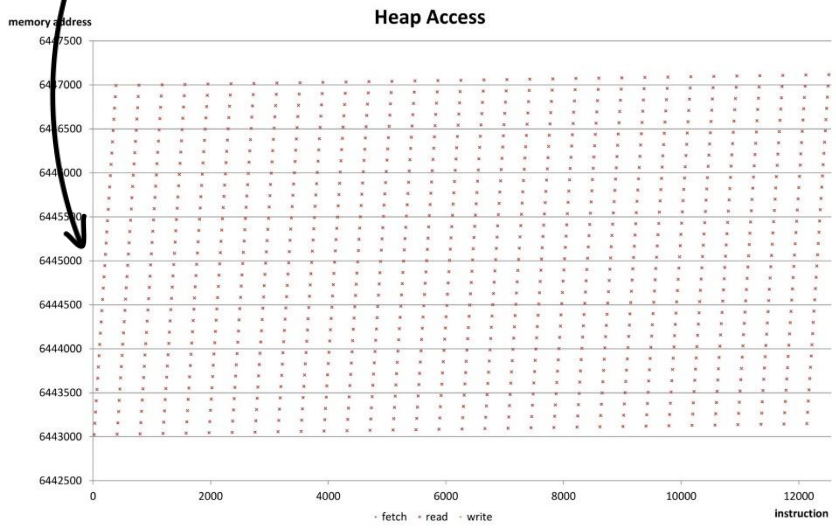
Locality: Memory Access Patterns

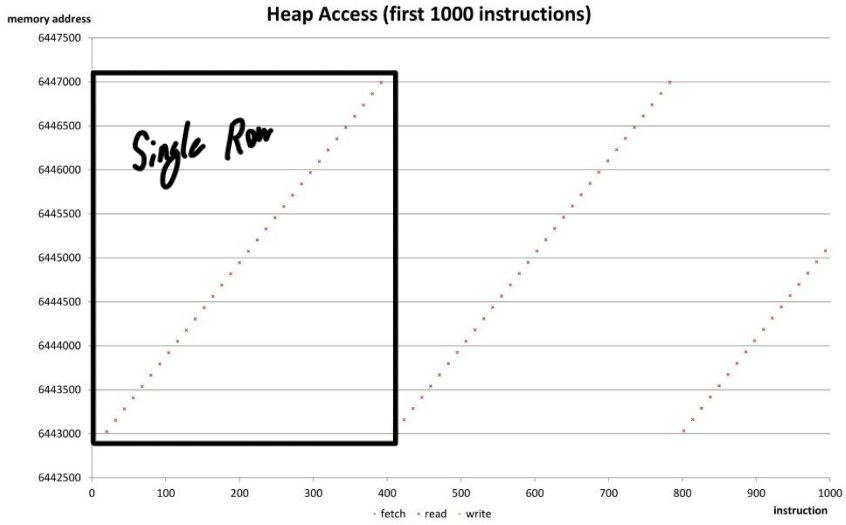
- array2d_sum_cm()





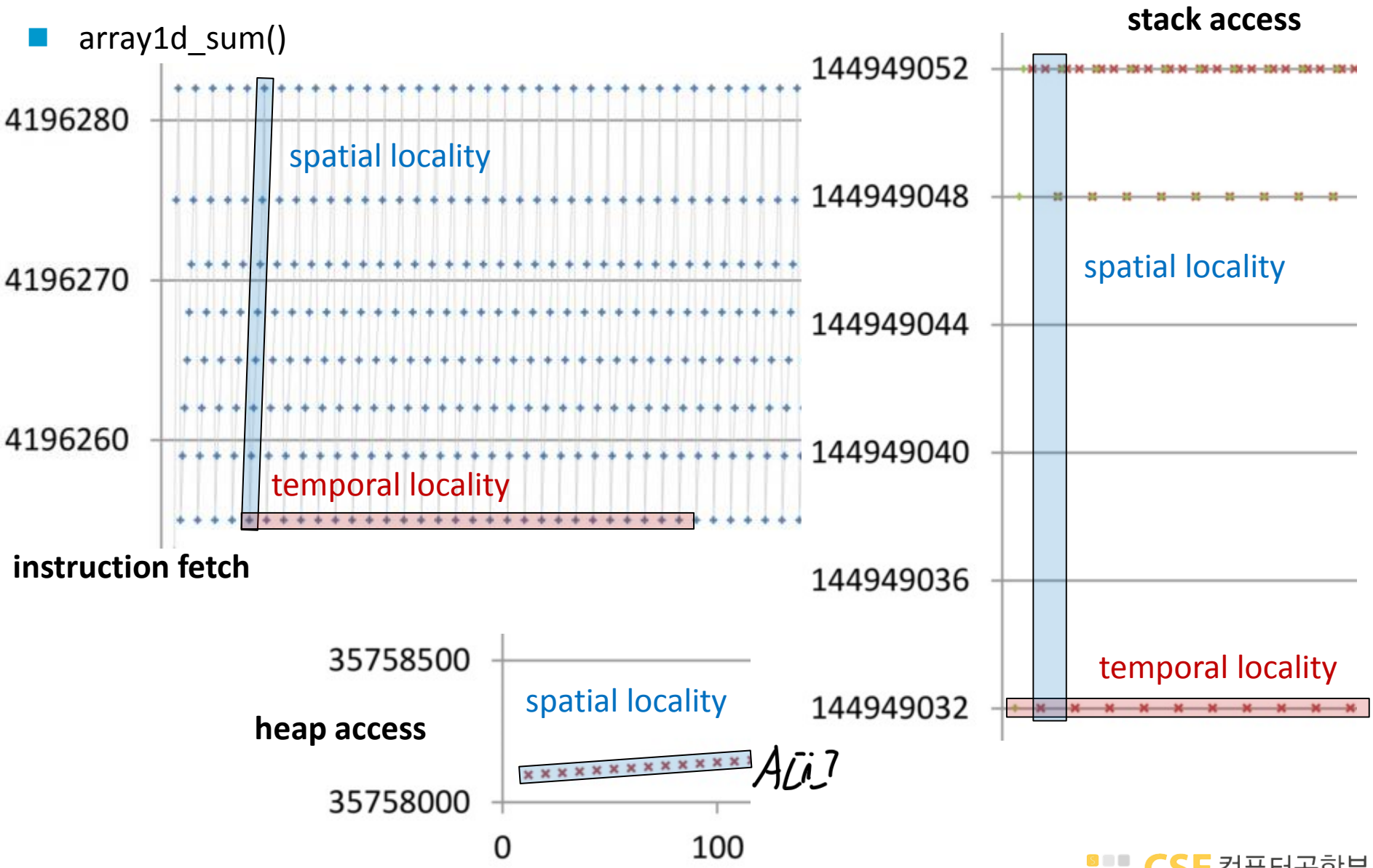
i and j





Locality: Memory Access Patterns

- array1d_sum()



The Solution: Memory Hierarchies

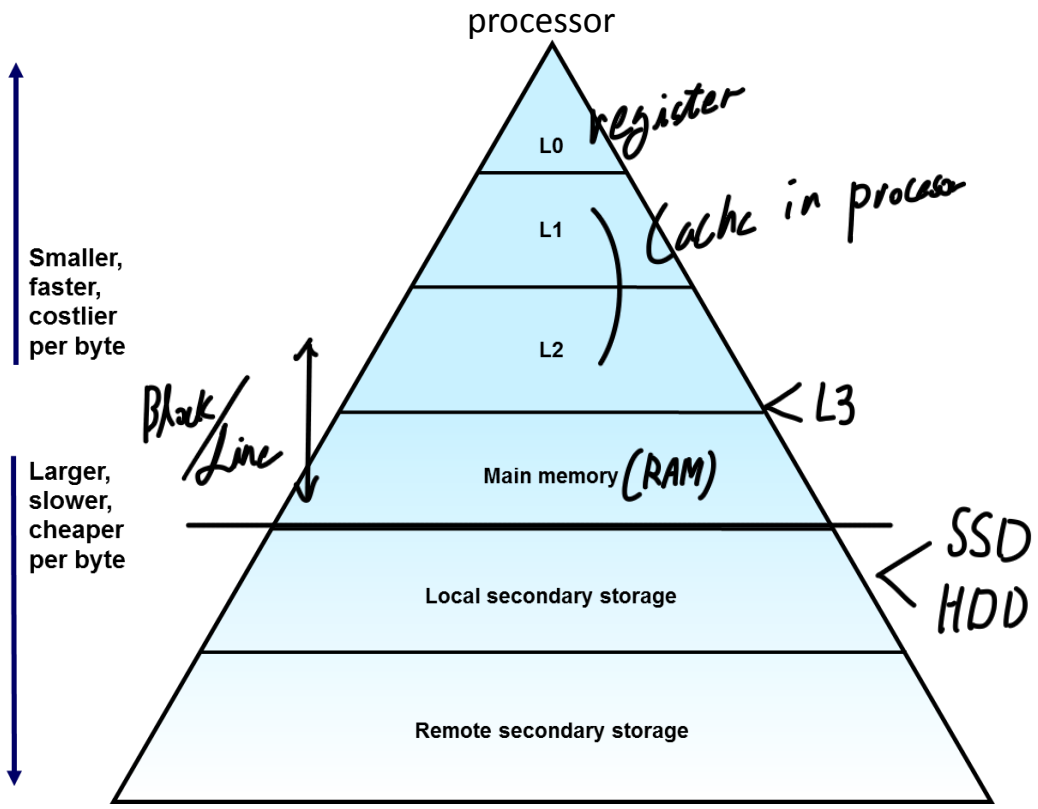
- Some fundamental and enduring properties of hardware and software:
  - Fast storage technologies cost more per byte, have less capacity, and require more power (heat!).
  - The gap between CPU and main memory speed is widening.
  - Well-written programs tend to exhibit good locality.

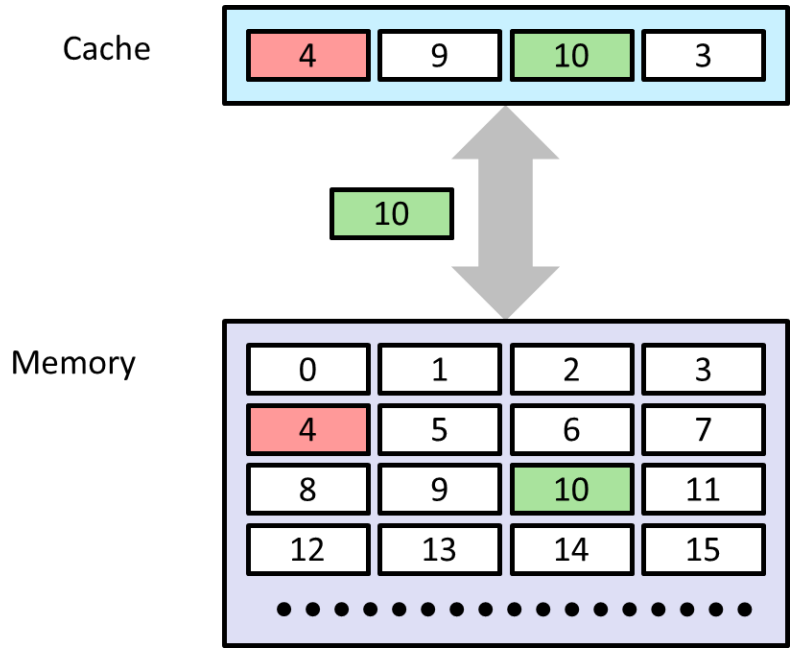
- These fundamental properties complement each other beautifully.

- They suggest an approach for organizing memory and storage systems known as a ***memory hierarchy***.

The Solution: Memory Hierarchies

> Ideally one would desire an indefinitely large memory capacity such that any particular … word would be immediately available … We are … forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.
>
> Burks, Goldstine, and von Neumann,1946





Caching in the Memory Hierarchy

Cache Memories

- From the Merriam-Webster Dictionary:

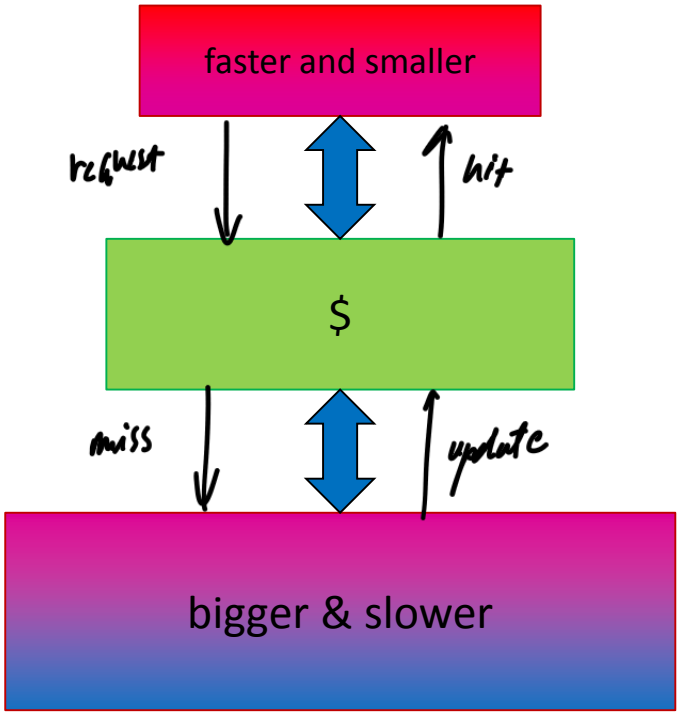
  cache:
  1 a: a hiding place especially for concealing and preserving provisions or implements
  b: a secure place of storage

  2 : something hidden or stored in a cache

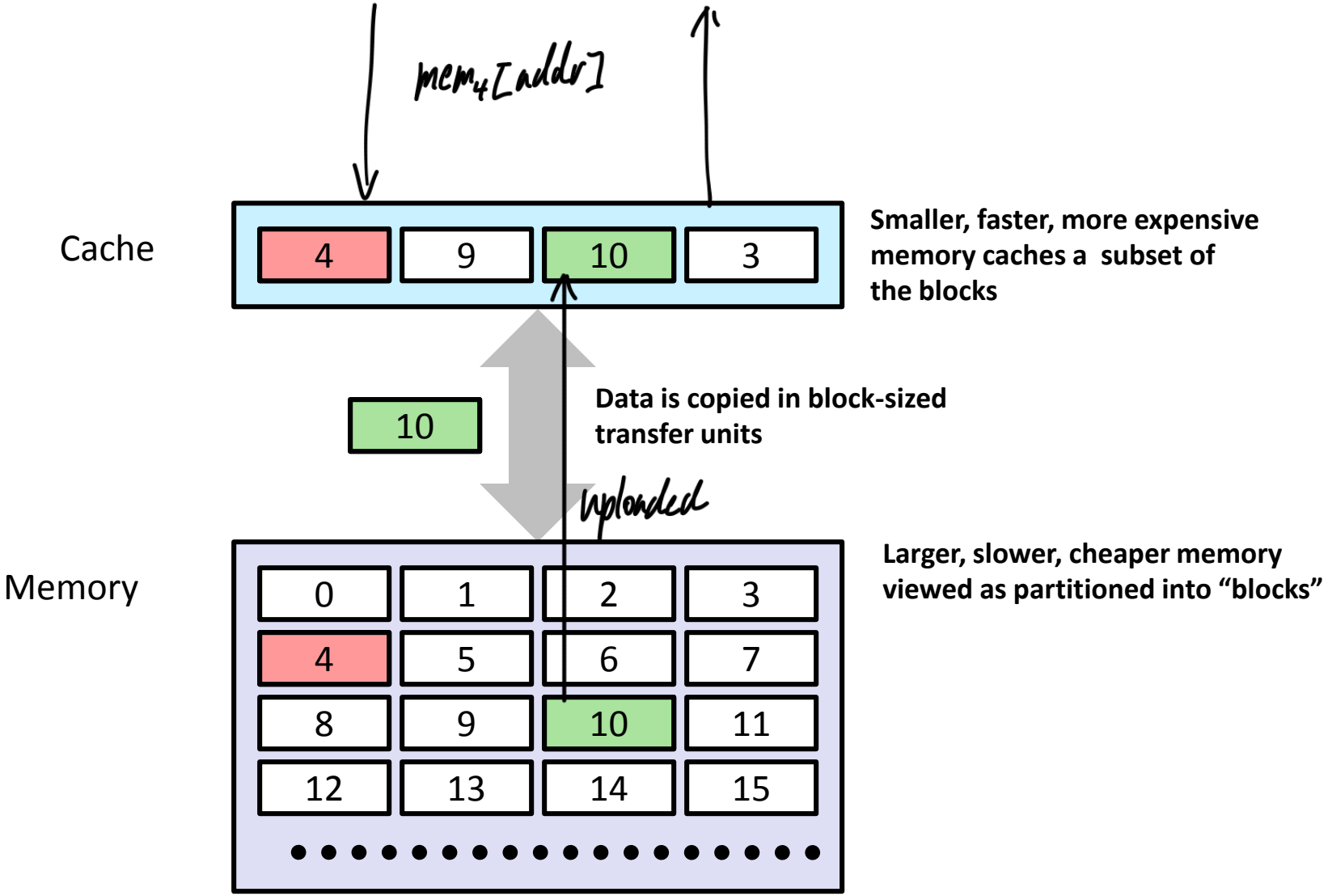
  3 : a computer memory with very short access time used for storage of frequently or recently used instructions or data

Cache Memories

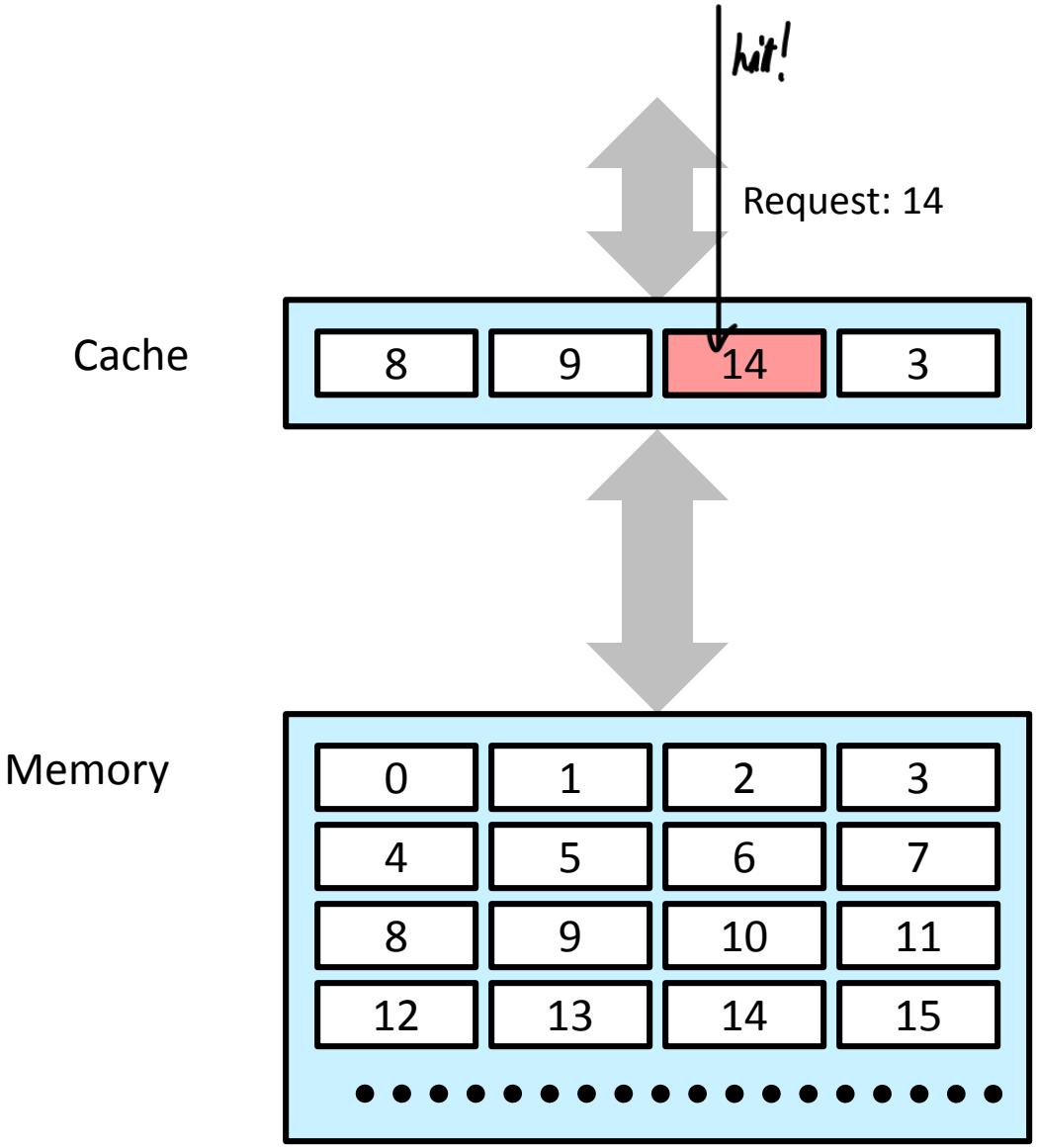
- **Hardware caches**
  hardware-managed memories that transparently buffer frequently or recently used data from a lower level in the memory hierarchy in order to provide faster access to the upper level
  - usually built from SRAM
  - operation
    - upper level requests datum ***d*** at address ***a***
    - cache checks if it currently holds a copy of ***d***
    - if yes (*cache hit*): return datum ***d***
    - if no (*cache miss*):
      request datum ***d*** at address ***a*** from lower level
      then
      - return datum ***d*** to upper level
      - store datum ***d*** in cache



General Concepts



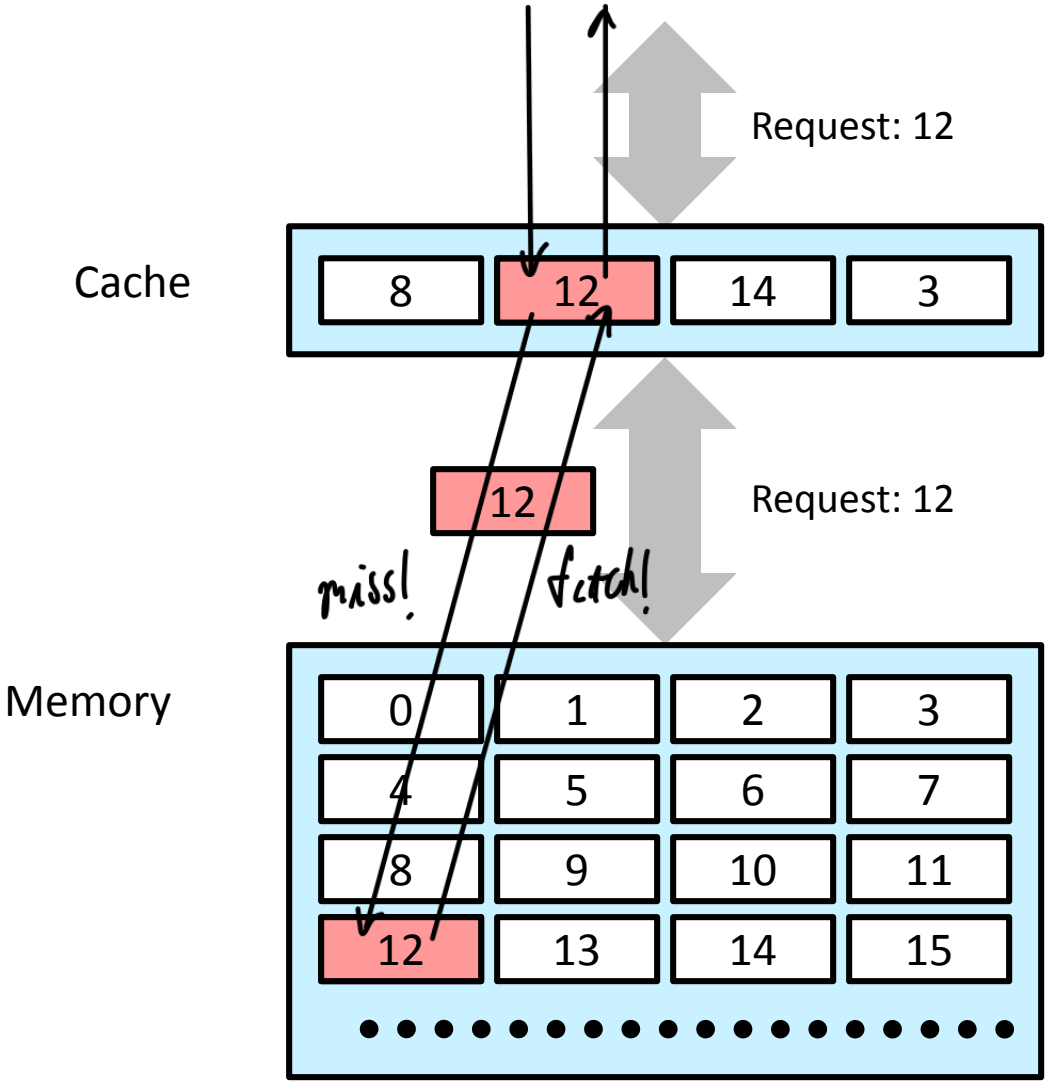
General Concepts: Hit



Data in block b is needed

Block b is in cache:
Hit!

General Concepts: Miss



Data in block b is needed

Block b is not in cache:
Miss!

Block b is fetched from *memory*

Block b is stored in cache

- Placement policy: determines where b goes
- Replacement policy: determines which block gets evicted (victim)

General Concepts: Types of Cache Misses

- **Cold (compulsory) miss**
  - Cold misses occur because the cache is empty.
- **Conflict miss** (set associative cache only) — kind of hash miss...
  - Most caches limit blocks at level k+1 to a small subset (sometimes a singleton) of the block positions at level k.
    - e.g. Block i at level k+1 must be placed in block (i mod 4) at level k.
  - Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block.
    - e.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.
- **Capacity miss**
  - Occurs when the set of active cache blocks (working set) is larger than the cache.

Examples of Caching in the Hierarchy

Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-8 bytes words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware
L1 cache	64-bytes block	On-Chip L1	1	Hardware
L2 cache	64-bytes block	On/Off-Chip L2	10	Hardware
Virtual Memory	4-KB page	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	AFS/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 Memories

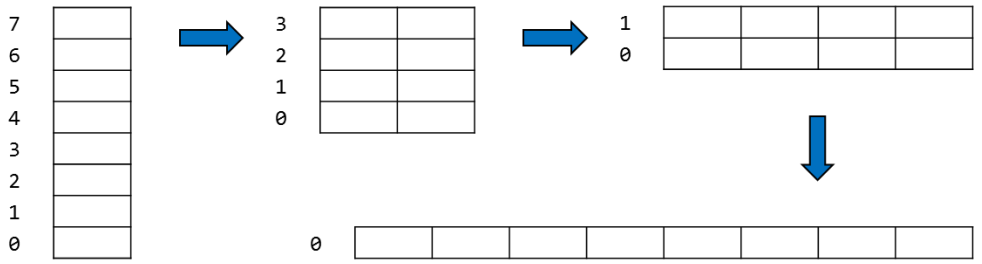
- **A cache implementation must provide answers to the following four questions**

1. Where can a block be placed?
   (**block placement policy**)

2. How is a block found?
   (**block identification policy**)

3. Which block should be replaced on a miss?
   (**block replacement policy**)
   (delete oldest hitted)

4. What happens on a write?
   (**write strategy**)



Cache Organization

Direct-mapped Cache : modulo

- **First approach: Direct-mapped cache**
  - cache with *c* bytes cache capacity
  - minimal storage unit is one block of size *b* bytes

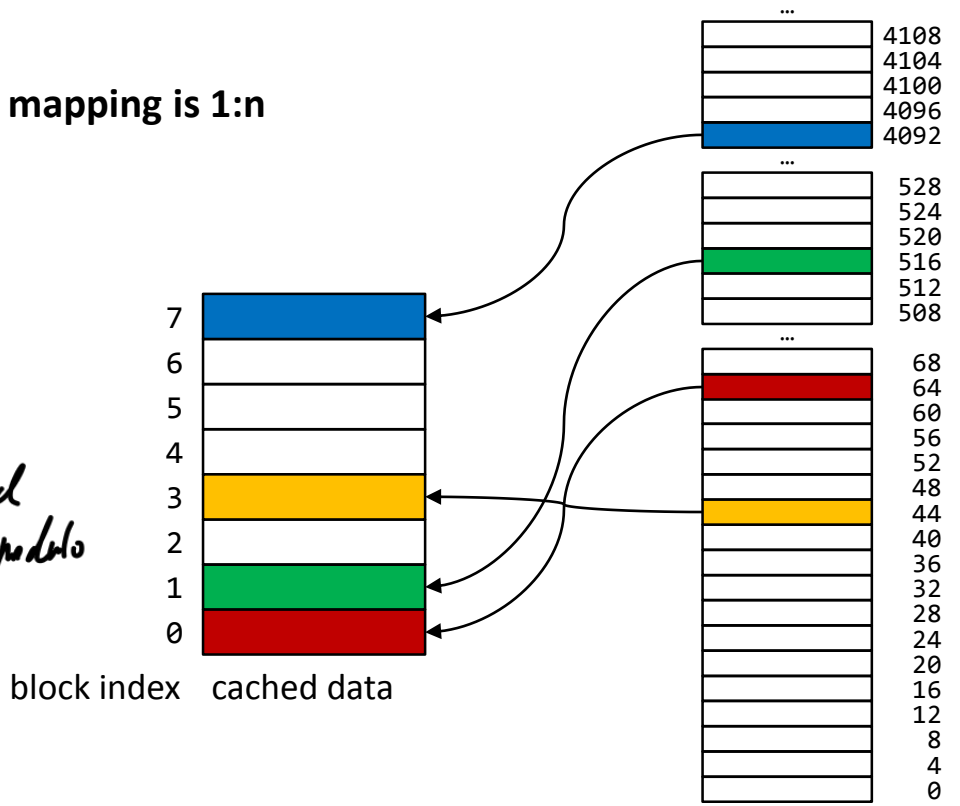
- **Problem 1: where should we store datum d?**
  Lower-level memory > cache capacity, i.e., the mapping is 1:n
  - block index *idx* for a datum *d* at address *a*

$$idx = (a/b) \% (c/b)$$

addr / b

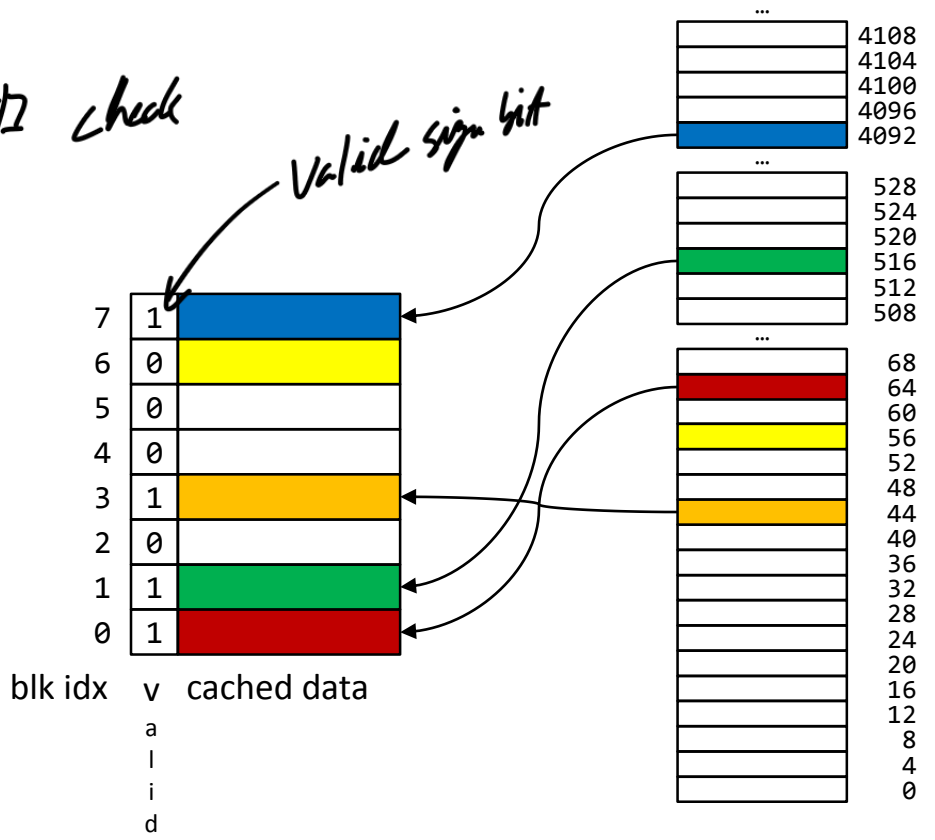
  - example: b = 4, c = 32
    - request: mem$_4$[516]
      → idx = (516/4) % (32/4) = 129 % 8 = 1
    - request: mem$_4$[4092]
      → idx = (4092/4) % (32/4) = 1023 % 8 = 7
    - request: mem$_4$[64]
      → idx = (64/4) % (32/4) = 16 % 8 = 0
    - request: mem$_4$[44]
      → idx = (44/4) % (32/4) = 11 % 8 = 3

hashed by modulo



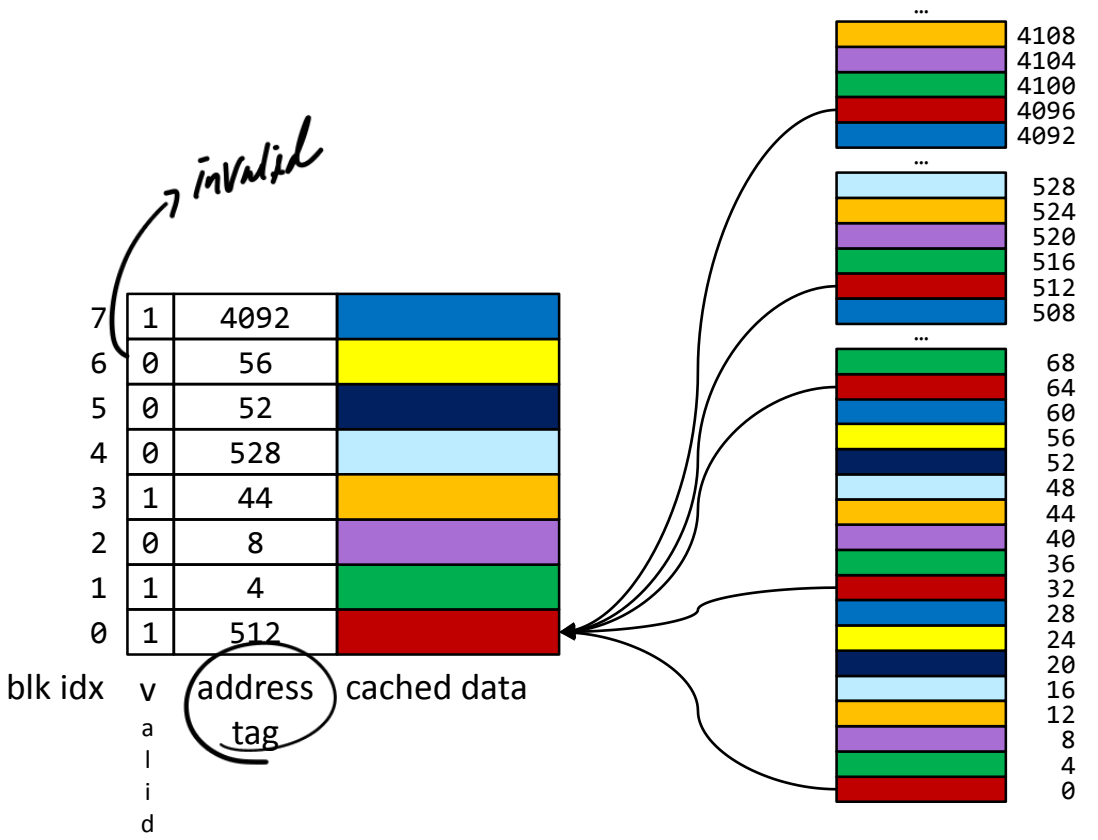
Direct-mapped Cache

- **Problem 2: how do we know whether a block in the cache is valid?**
  - add a valid bit for each block
    valid == 1: cache hit
    valid == 0: cache miss
    - mem_4[516] = hit
    - mem_4[64] = hit
    - mem_4[56] = miss



Direct-mapped Cache

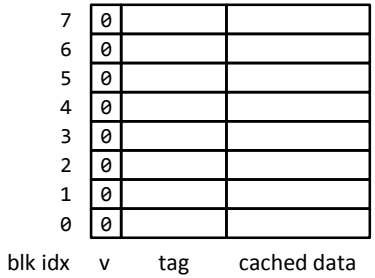
- **Problem 3: how do we know whether a valid block in the cache actually holds the requested datum?**
  - n addresses get mapped to the same cache block:
    - block index = 0 for 0, 32, 64, … = *i***c*
  - a valid bit alone is not sufficient, we also need to store the address of the datum in the cache
    - mem_4[516] = miss
    - mem_4[512] = hit
    - mem_4[56] = miss

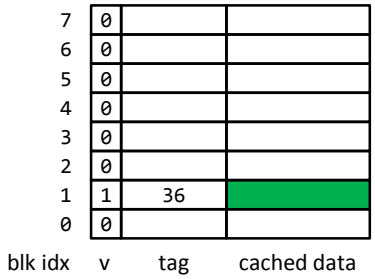


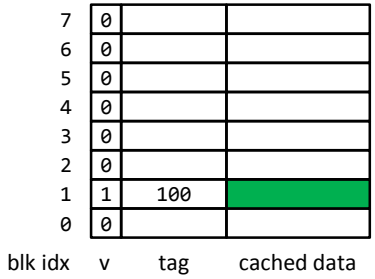
Direct-mapped Cache

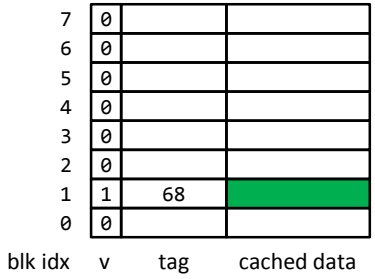
1:N matching!

- **How good is a direct-mapped cache?**
  - not bad for random accesses (uniform address case)
  - what about non-random accesses? (conflict miss)
    - start with an empty cache and assume our code repeatedly accesses addresses 36, 100, 68, 36, 100, 68, …
    - request: $mem_4[36]$ → block index = 1 → miss, place 36 in block 1
    - request: $mem_4[100]$ → block index = 1 → miss, place 100 in block 1
    - request: $mem_4[68]$ → block index = 1 → miss, place 68 in block 1
    - request: $mem_4[36]$ → block index = 1 → miss, place 36 in block 1
    - request: $mem_4[100]$ → block index = 1 → miss, place 100 in block 1
    - request: $mem_4[68]$ → block index = 1 → miss, place 68 in block 1
    - hmmm…







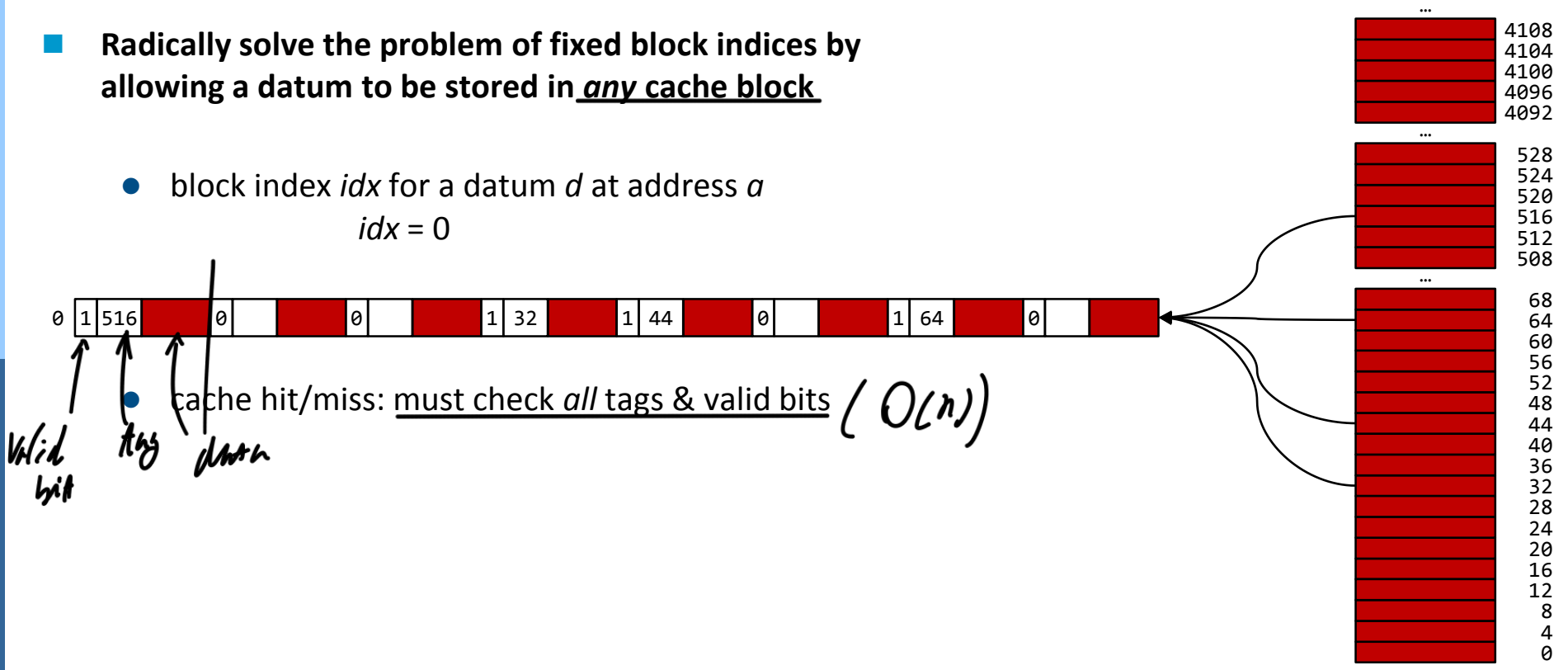


Fully-associative Cache

- **Second approach: Fully-associative cache**
  - cache with *c* bytes cache capacity
  - minimal storage unit is one block of size *b* bytes

- **Radically solve the problem of fixed block indices by allowing a datum to be stored in *any* cache block**
  - block index *idx* for a datum *d* at address *a*

    idx = 0
  - cache hit/miss: must check *all* tags & valid bits



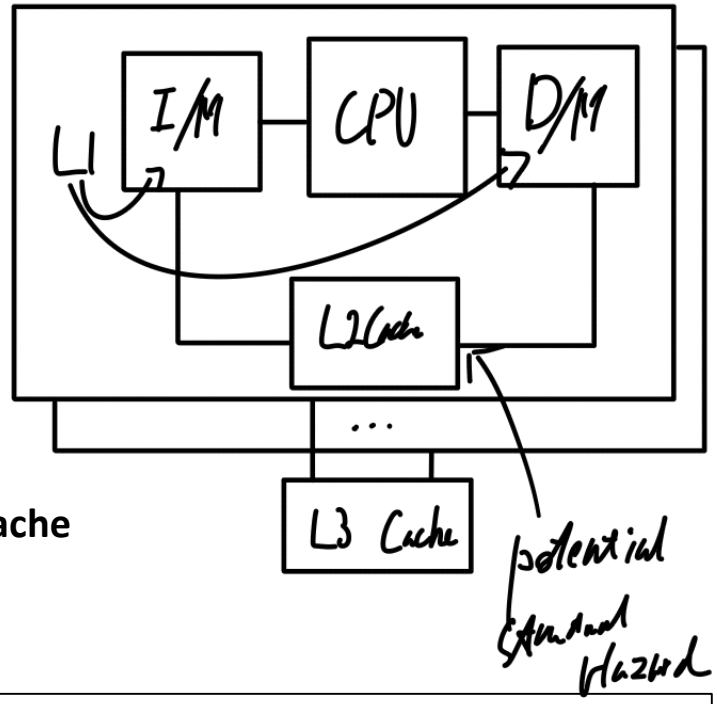
Fully-associative Cache

- **Block replacement**
  - assume that cache is fully occupied

and a cache miss occurs. Which block should be overwritten?

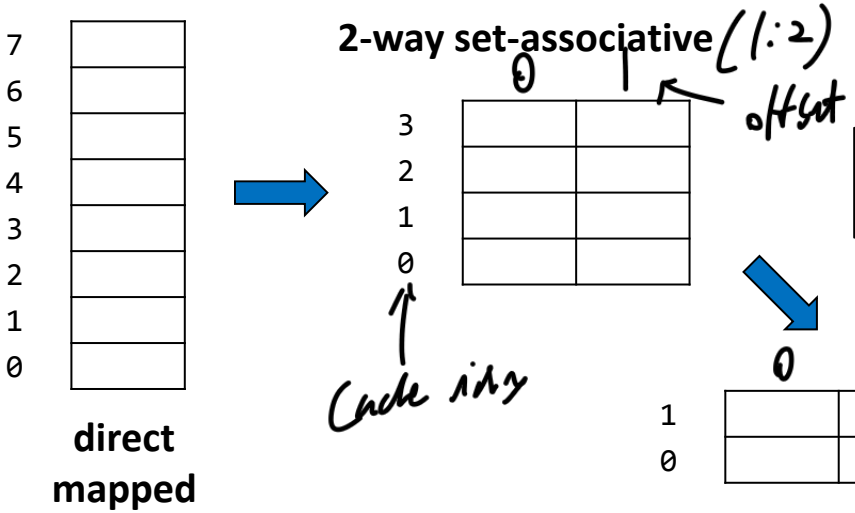
  - **Block replacement policy**
    - optimal: Bélády's algorithm → what will it?
    - practical are:
      - random
      - LRU (least recently used) : double linked list / queue
      - pseudo-LRU → hard to implement in hardware + O(n) time complexity
  - disadvantages of fully-associative caches
    - complex, slow(er), high energy consumption

N-Way Set-Associative Cache

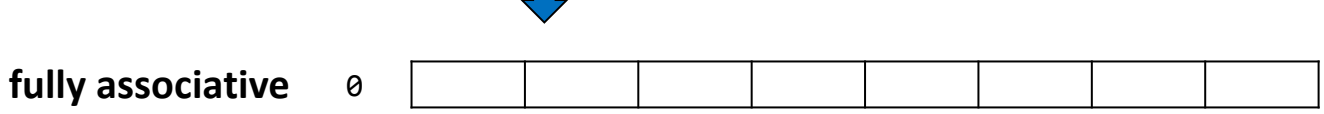


- **General approach: N-way set-associative cache**
  - cache with *c* bytes cache capacity
  - minimal storage unit is one block of size *b* bytes

- **Hybrid between a direct-mapped and a fully-associative cache**

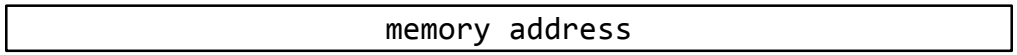


w ways: block index *idx* for a datum *d* at address *a:*
idx = (*a*/b) % (*c*/*b*/*w*)

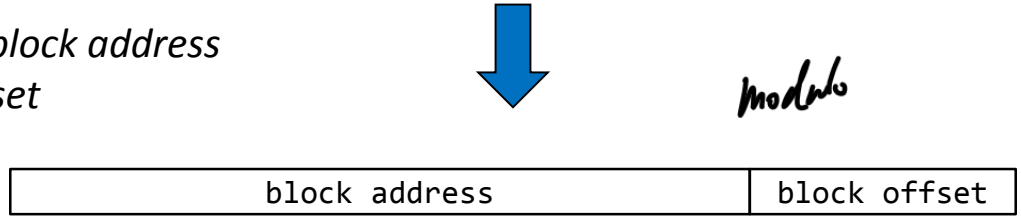


N-Way Set-Associative Cache

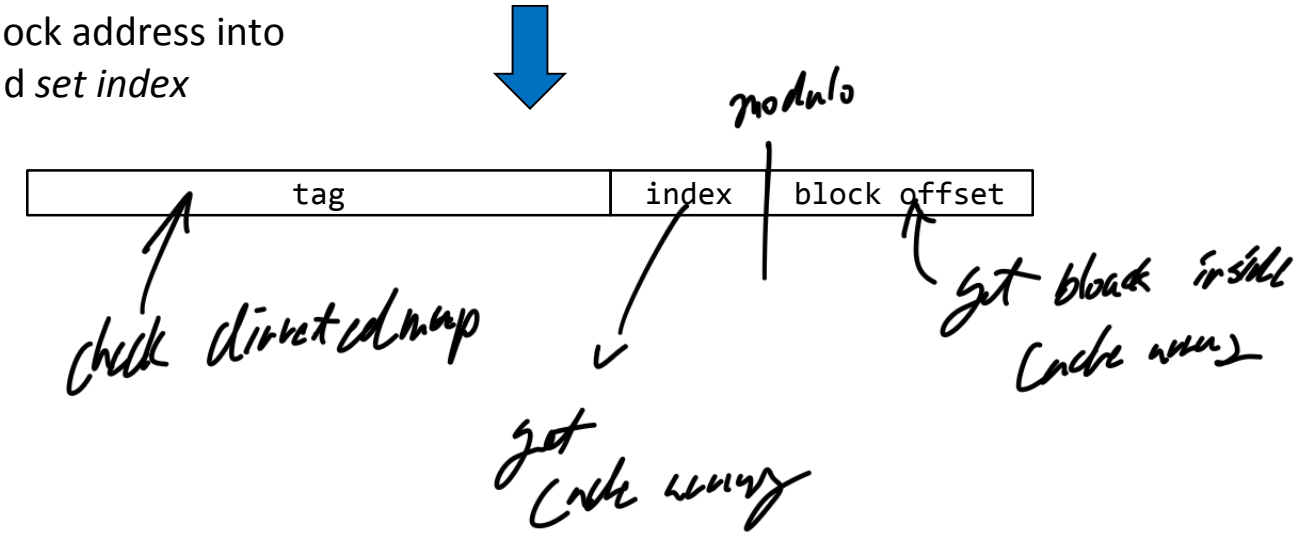
- **Division of an address**
  - full address



  - division into *block address* and *block offset*

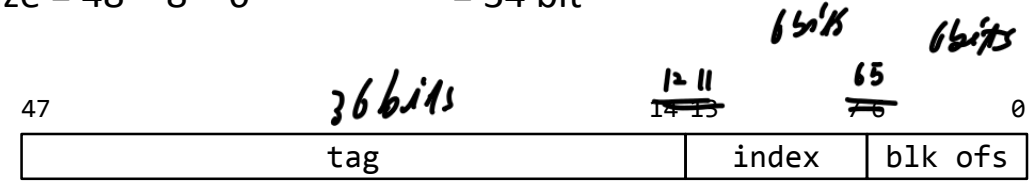


  - division of block address into *cache tag* and *set index*



N-Way Set-Associative Cache

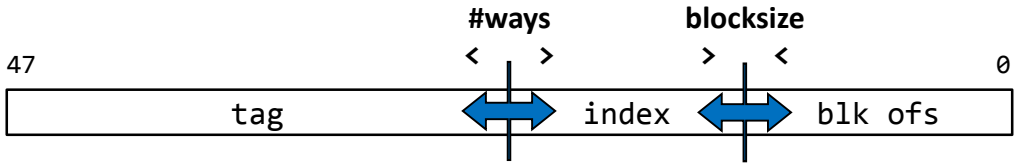
- **Address division examples**
  - full address: 48 bit
  - capacity: 32KB, 2-way set associative, block size: 64 bytes
    - block offset: 0 ~ 63 → 6 bit
    - # of sets = 32K / 64 / 2 = 256 → 8 bit
    - tag size = 48 – 8 – 6 = 34 bit



    - tag overhead: # blocks * tagsize = 32K / 64 * 34 bit = 2’176 B (6.25%)
  - capacity: 32KB, 8-way set associative, block size: 64 bytes
    - # of sets = 32K / 64 / 8 = 64 → 6 bit
    - tag size = 48 – 6 – 6 = 36 bit
    - tag overhead: # blocks * tagsize = 32K / 64 * 36 bit = 2’304 B (7.03%)

N-Way Set-Associative Cache

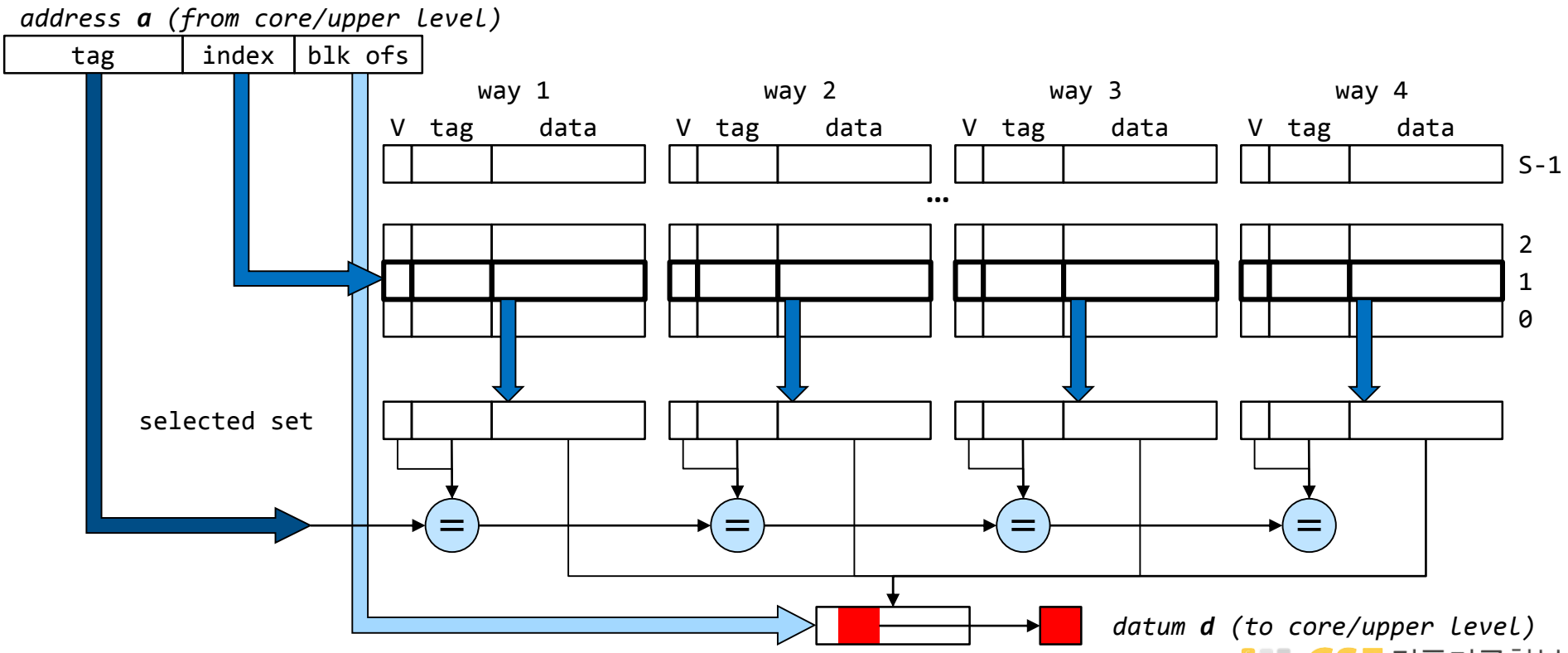
- **Address division examples**
  - full address: 48 bit
  - capacity: 32KB, fully associative, block size: 64 bytes
    - block offset: 0 ~ 63 → 6 bit
    - # of sets = 1 → 0 bit
    - tag size = 48 – 0 – 6 = 42 bit
    - tag overhead: # blocks * tagsize = 32K / 64 * 42 bit = 2’688 B (8.20%)
  - effect on associativity and blocksize on address division



N-Way Set-Associative Cache

- **Block placement/identification**
  - computation of set s = (a/b) % (c/b/w)
  - hit/miss: check the valid bits and compare tag of all blocks in a set with the address tag

b: size of block
w: n-way
c: cache capacity
number of sets = index size
(tag + index) % number of sets = index



N-Way Set-Associative Cache

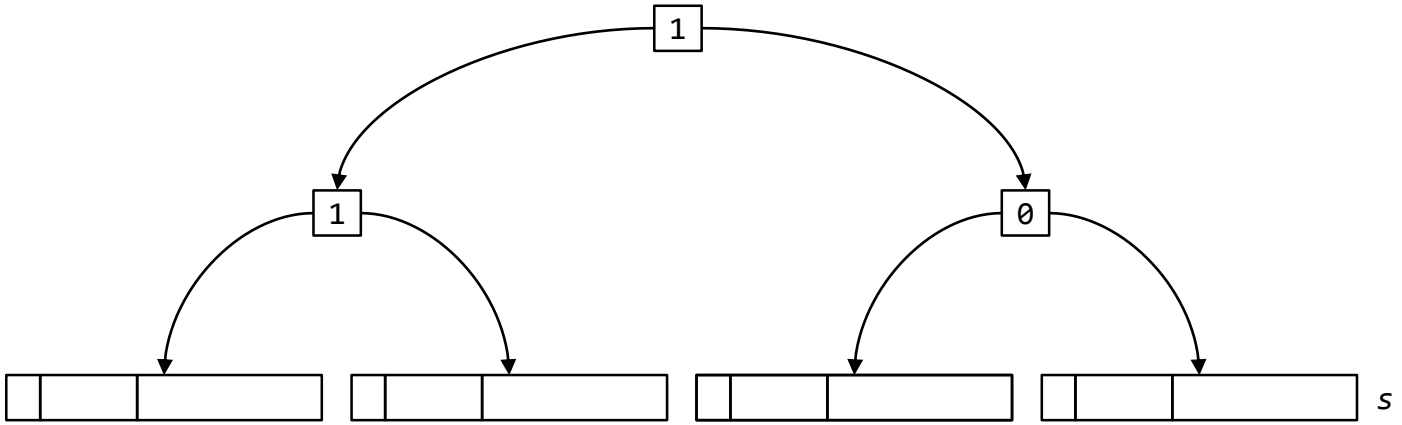
- **Block replacement**
  - situation: new block fetched after a cache miss must be written into one of the n ways of the appropriate set.
  - if there are free blocks, pick one of them
  - if no blocks are free in the set, select a victim block to evict

- **Block replacement (selection) strategies**
  - (pseudo)random
  - LRU
    - exact LRU difficult in hardware with >4 ways → use pseudo-LRU
  - MRU
  - LFU (least-frequently-used)
  - …

Tree Pseudo-LRU Replacement

- Maintaining exact LRU data structures in hardware is difficult and has a high overhead

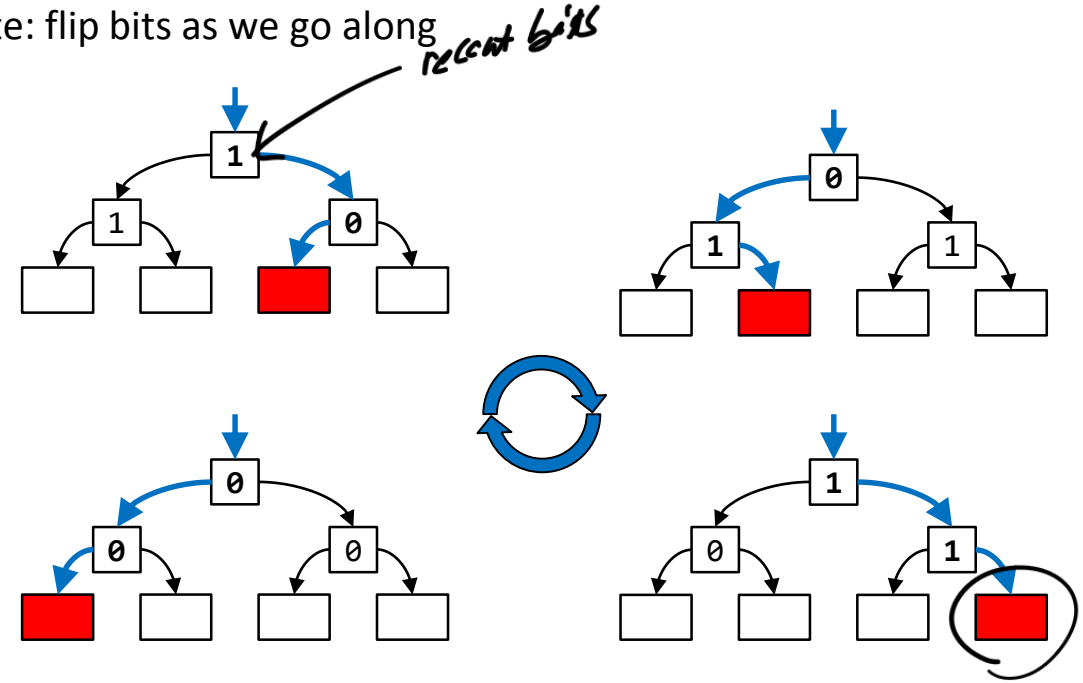
- Tree Pseudo-LRU approximates LRU and uses only one additional bit per cache block
  - ways in a set form the leaves of a binary tree
  - each non-leaf node has one direction bit d. ḋ=0: go left, d=1: go right



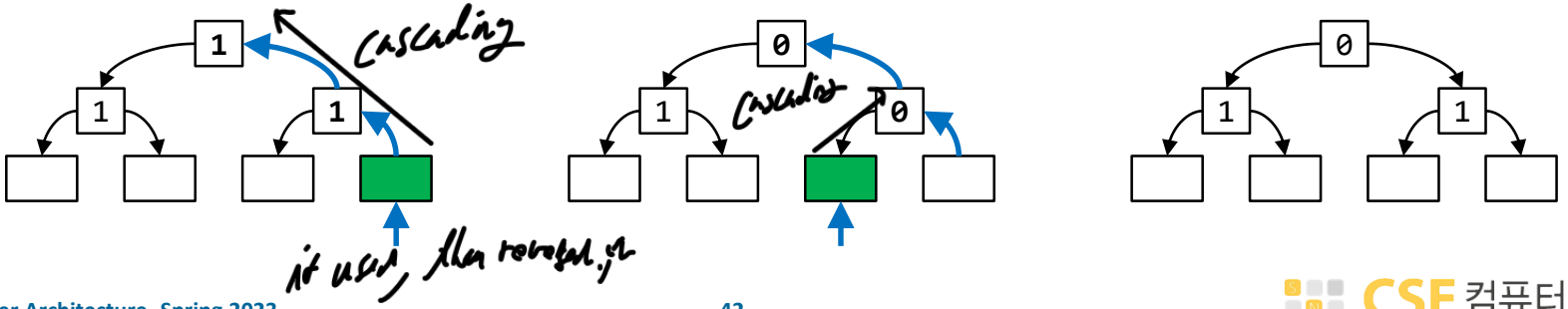
Tree Pseudo-LRU Replacement

direction bit
d=0: go left
d=1: go right

- Victim search: follow tree from root
  - with update: flip bits as we go along



  - cache access: follow tree up to root, set bits to point away from leaf



Write Strategies : when do we bring change to memory?

- **Write-through vs. write-back**
  - determines what happens on a memory write to a block currently in the cache
  - write-through: write modified value to next level in memory hierarchy
  - write-back: only modify cache line, do not send to next level
    - requires a dirty bit to keep track of modified blocks : to record updated
    - upon replacement, dirty blocks need to be written back to the next level

- **Write-allocate vs. no-write-allocate**
  - determines what happens on a write miss
  - write-allocate (aka fetch-on-write): block is loaded into cache, then modified
  - no-write-allocate (aka write around): modifications are sent to lower level and not cached

- Typically, write-through is used with no-write-allocate and write-back with write-allocate.
- To reduce the latency of writes, caches often have a write buffer
- Assuming the data is in the cache, why are writes typically slower than reads?

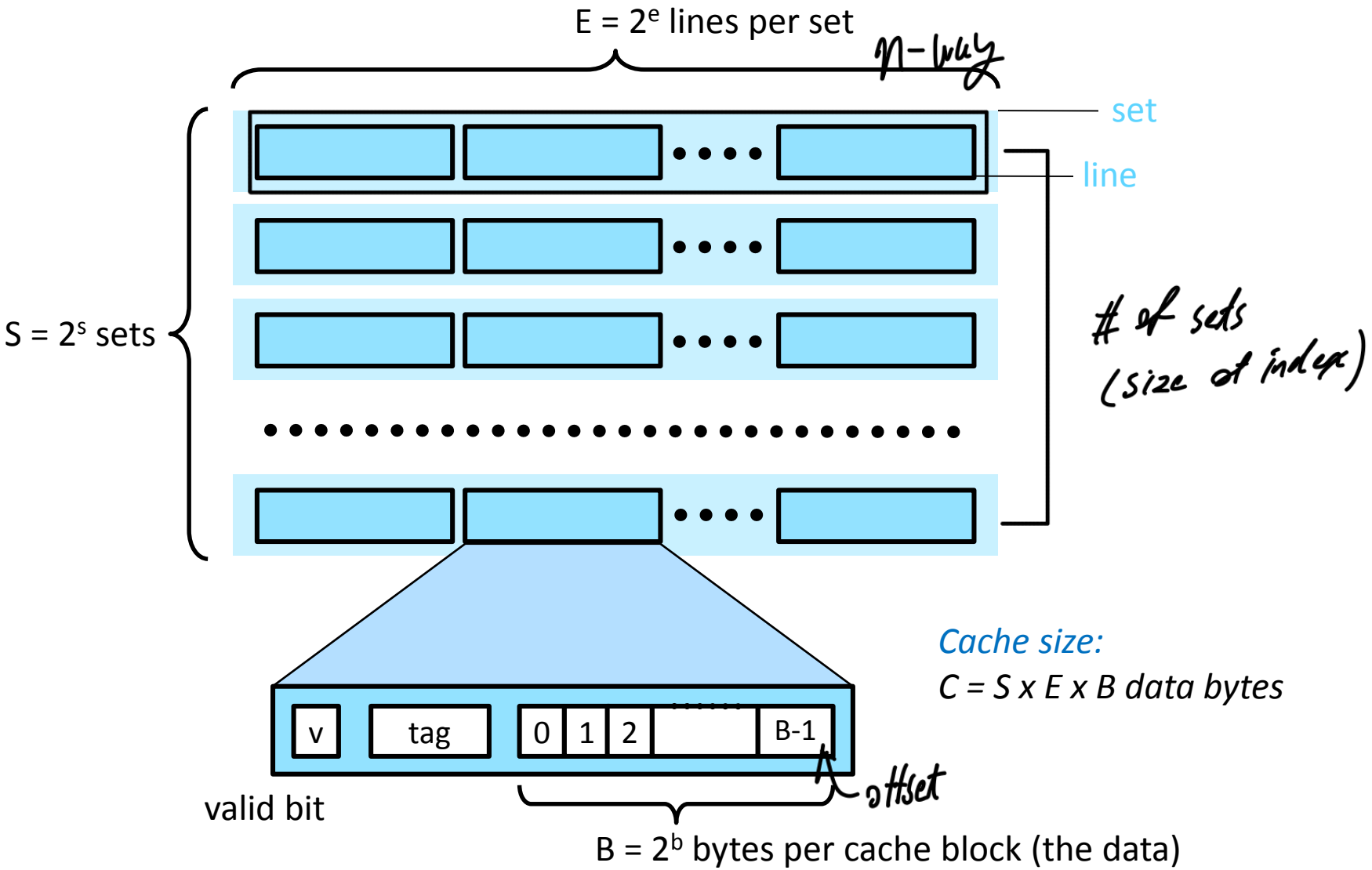
Classification of Cache Misses

- **Compulsory miss** (cold miss)
  - the very first access to a block cannot be a miss
  - example: empty cache, access address *a*

- **Capacity miss**
  - occur when the number of cache blocks a program accesses (maybe in a loop) is bigger than the number of blocks in the cache, i.e., the cache cannot contain all the data a program accesses
  - example: program accesses blocks 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, but because the cache can only hold 4 blocks there will be more than the 5 compulsory misses

- **Conflict miss**
  - caused by block replacements in a direct-mapped or set-associative cache
  - example: direct-mapped cache with 8 blocks, program accesses blocks 0, 8, 16, 32.

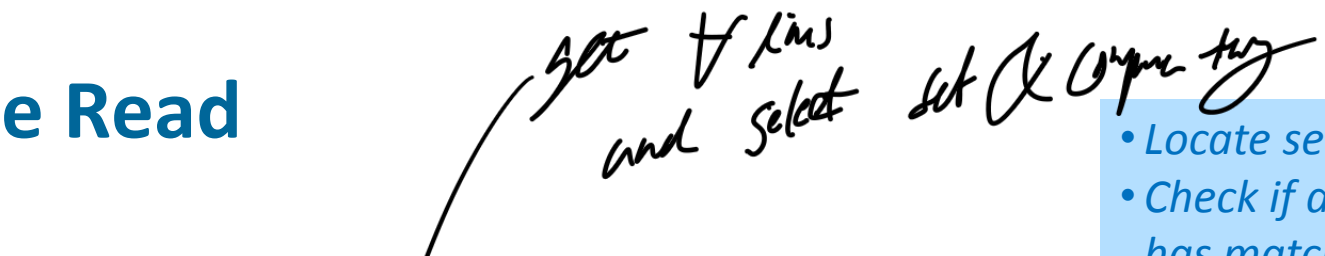
General Cache Organization (S, E, B)



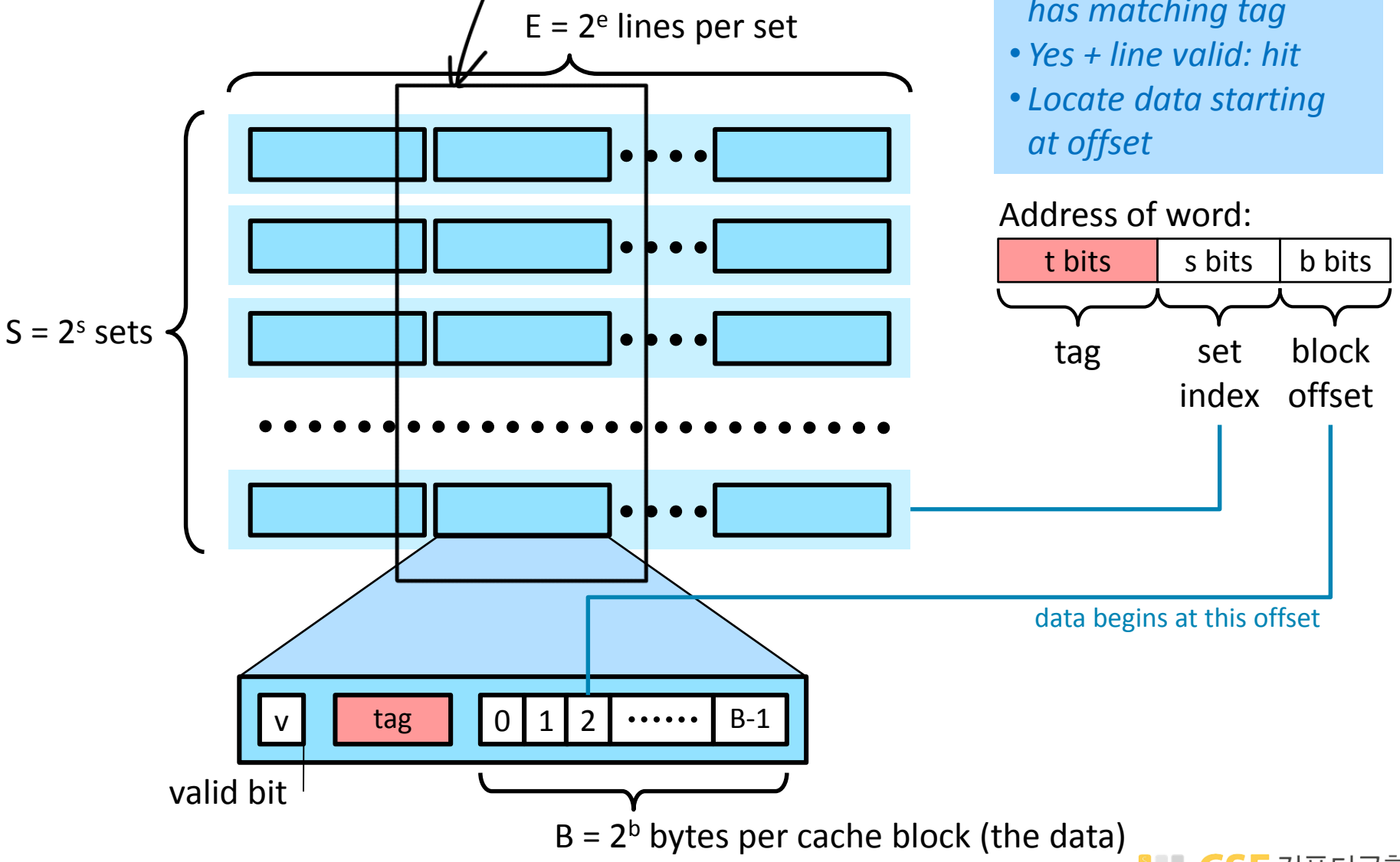
Cache size:

C = S x E x B data bytes

Cache Read



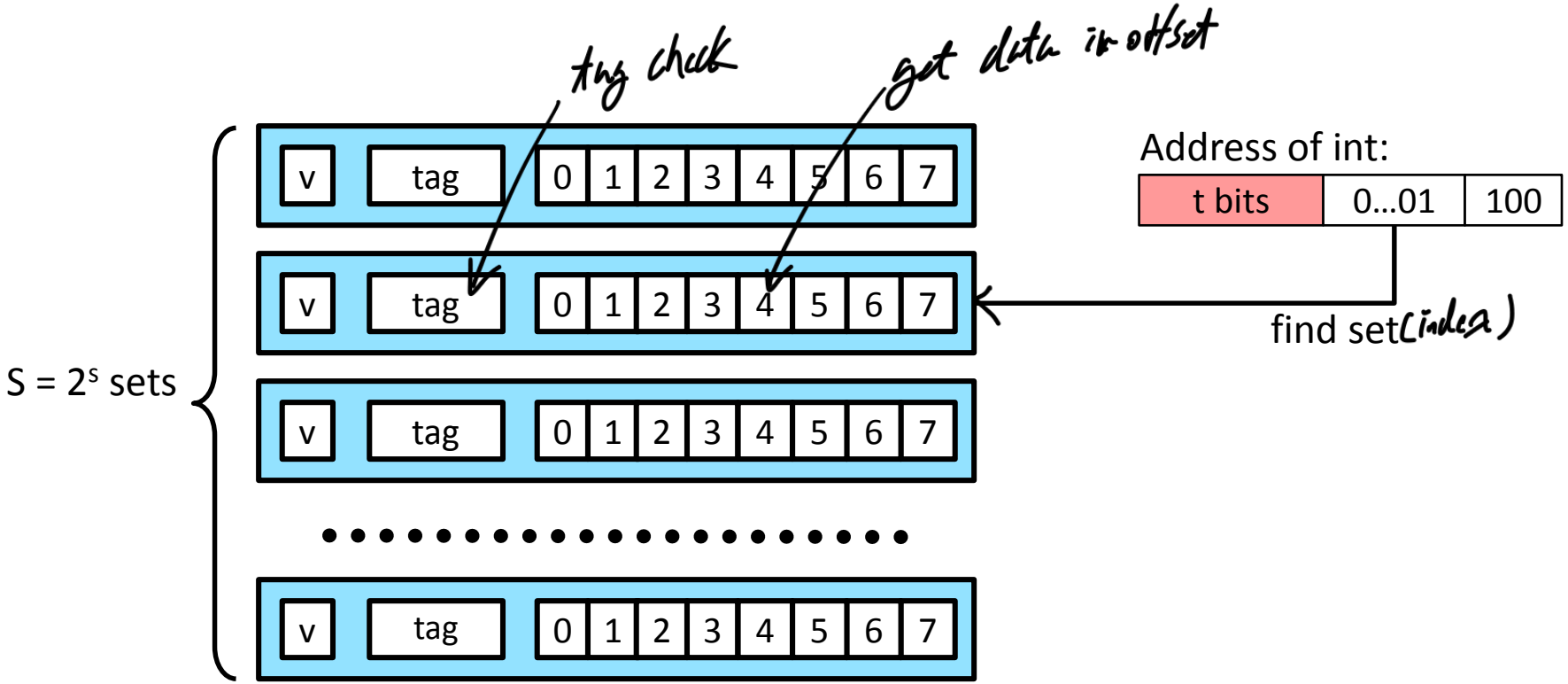
- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*



Example: Direct Mapped Cache (E = 1)

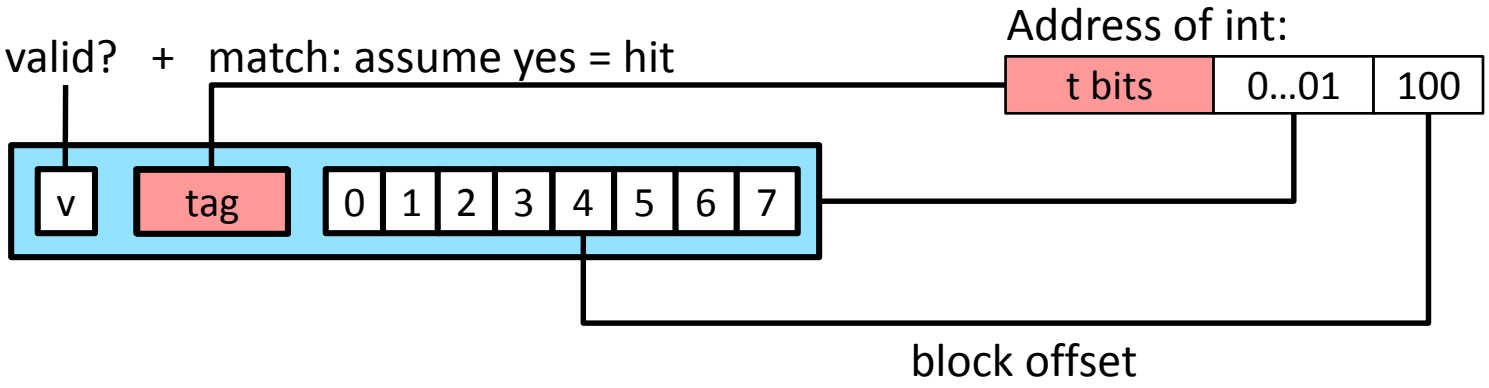
Direct mapped: One line per set
Assume: cache block size 8 bytes

tag index offset



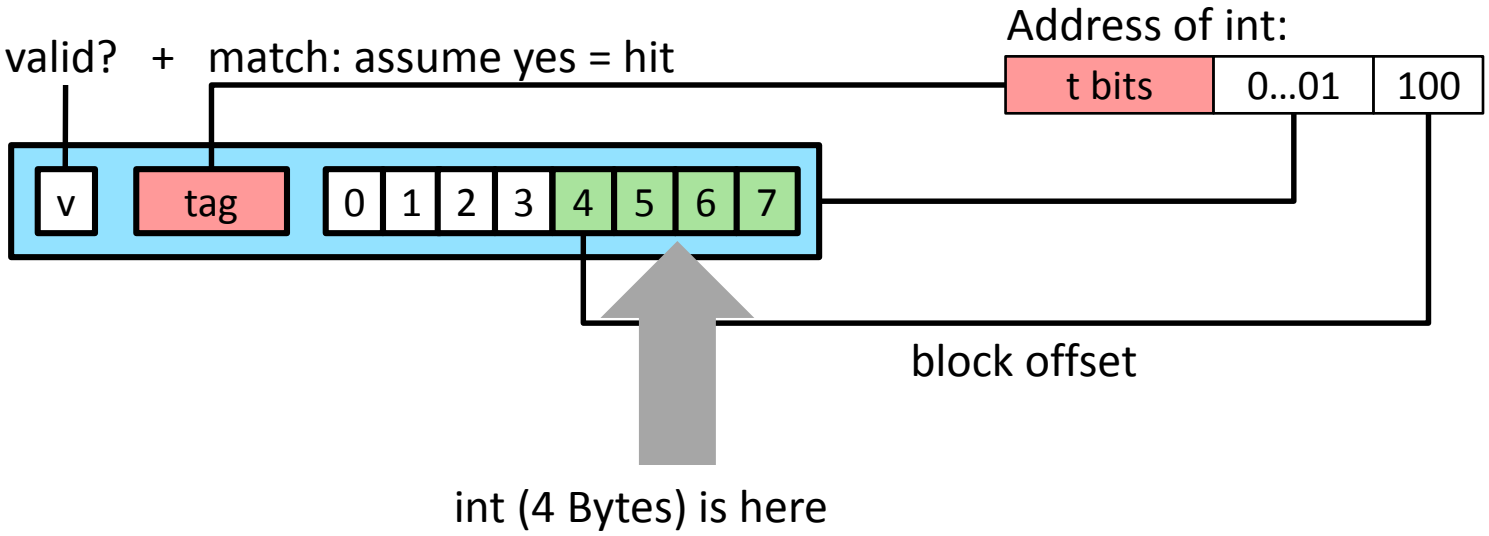
Example: Direct Mapped Cache (E = 1)

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



No match: old line is evicted and replaced

Direct-Mapped Cache Simulation

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

M=16 byte addresses, B=2 bytes/block,
S=4 sets, E=1 Blocks/set

Address trace (reads, one byte per read):

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

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

A Higher Level Example

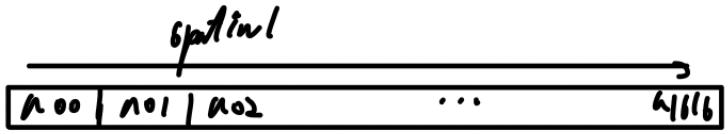
Ignore the variables sum, i, j

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

assume: cold (empty) cache,
a[0][0] goes here

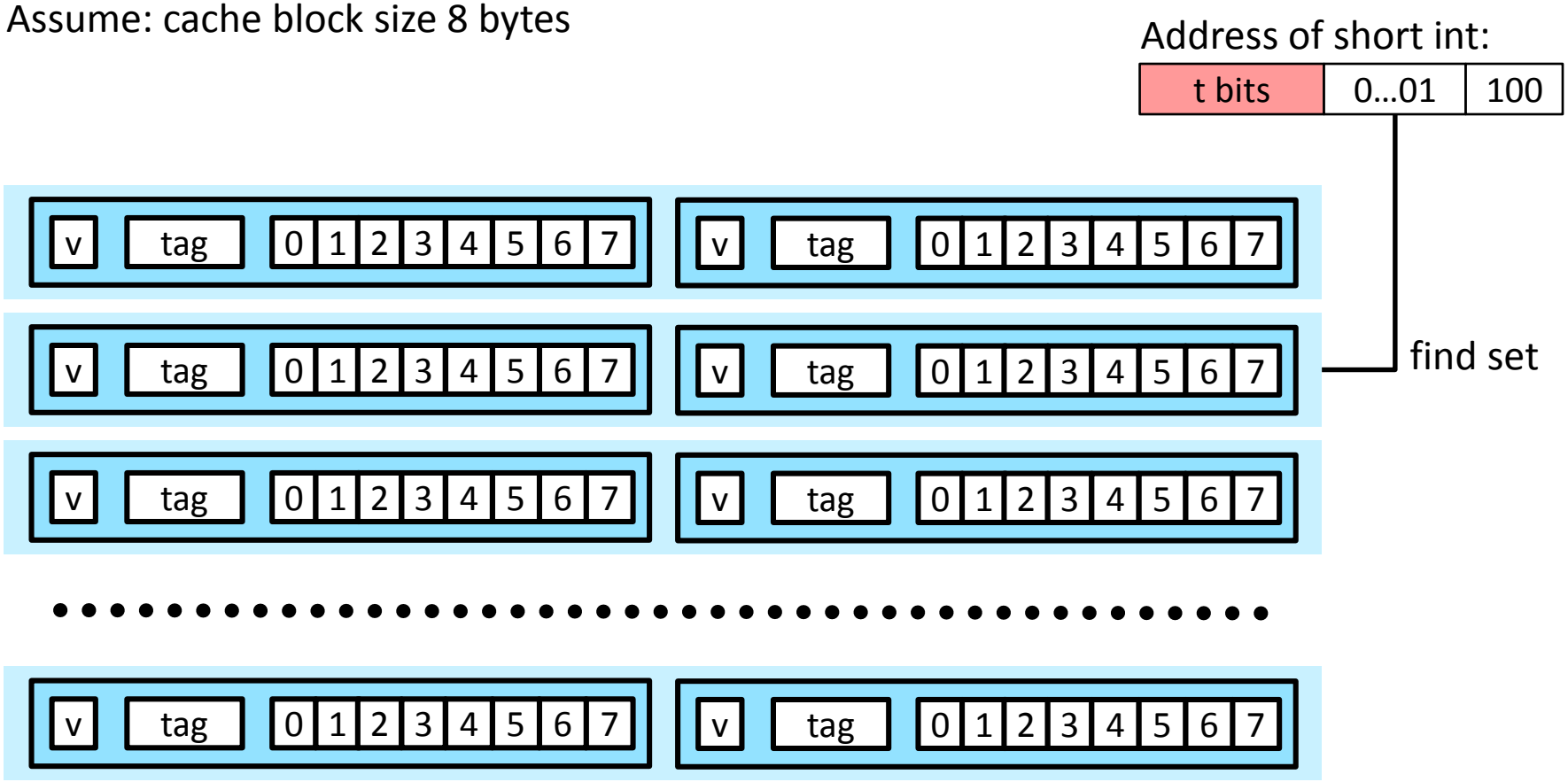
32 B = 4 doubles



E-way Set Associative Cache

E = 2: Two lines per set

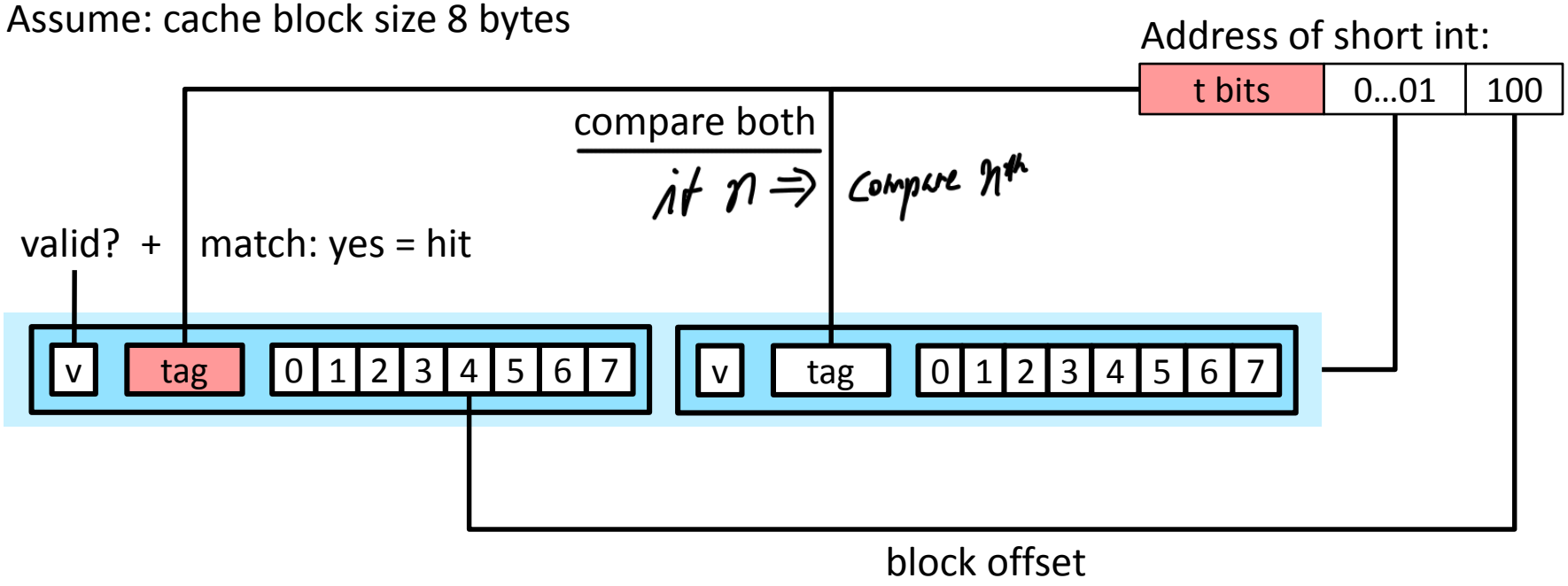
Assume: cache block size 8 bytes



E-way Set Associative Cache

E = 2: Two lines per set

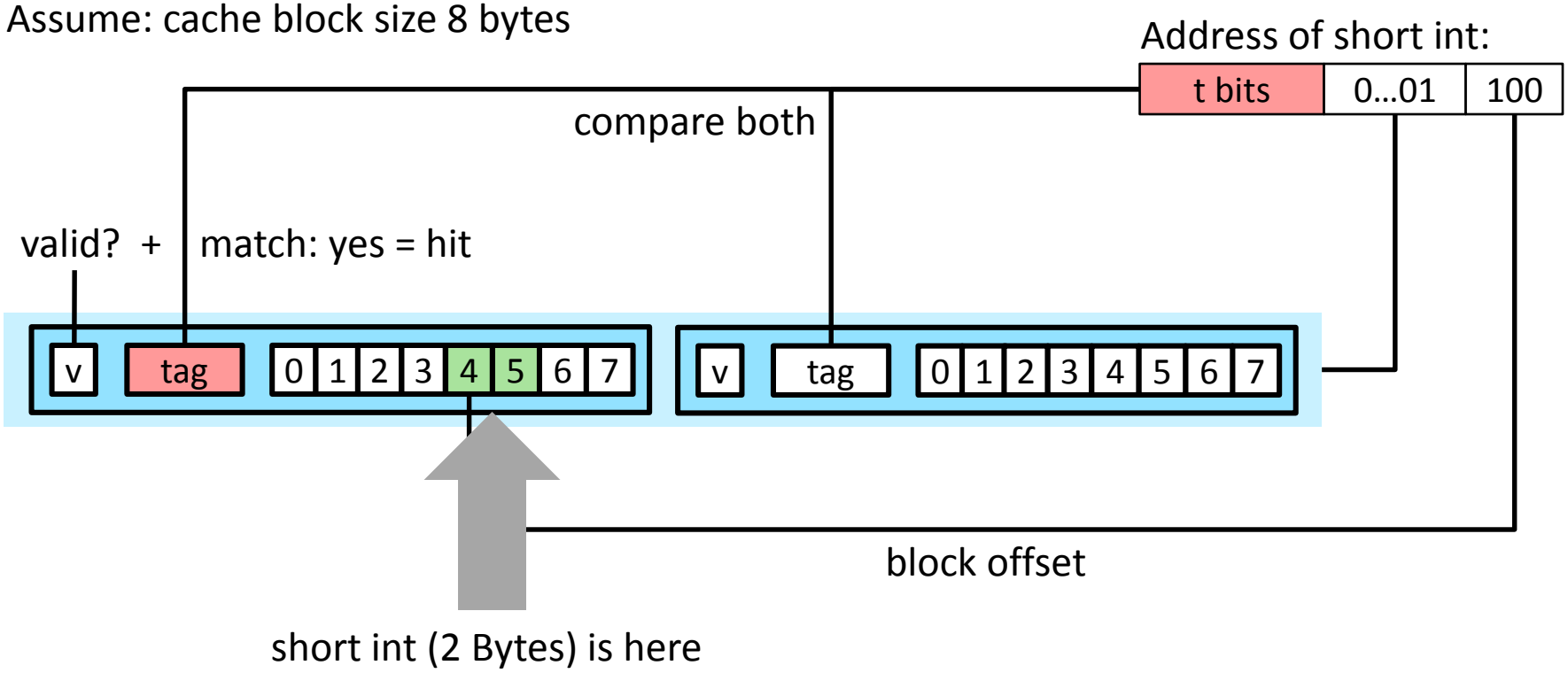
Assume: cache block size 8 bytes



E-way Set Associative Cache

E = 2: Two lines per set

Assume: cache block size 8 bytes



No match:

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

2-Way Set Associative Cache Simulation

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

M=16 byte addresses, B=2 bytes/block,
S=2 sets, E=2 blocks/set

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		

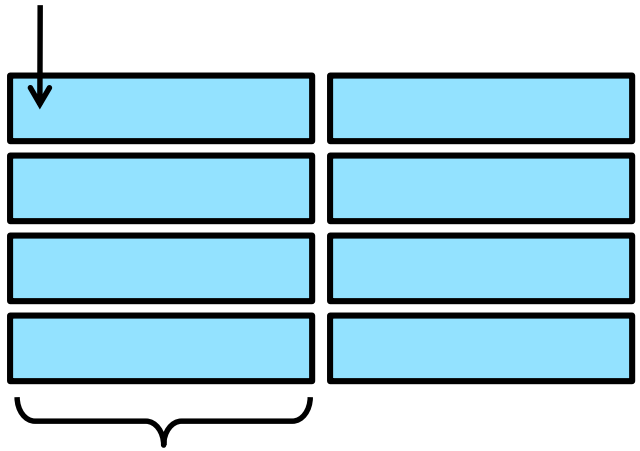
A Higher Level Example

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

Ignore the variables sum, i, j

assume: cold (empty) cache,
a[0][0] goes here



What about writes?

- Multiple copies of data exist:
  - L1, L2, Main Memory, Disk
- What to do on a write-hit?
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until replacement of line)
    - Need a dirty bit (line different from memory or not)
- What to do on a write-miss?
  - Write-allocate (load into cache, update line in cache)
    - Good if more writes to the location follow
  - No-write-allocate (writes immediately to memory)
- Typical
  - Write-through + No-write-allocate
  - **Write-back + Write-allocate**

A Common Framework for Memory Hierarchies

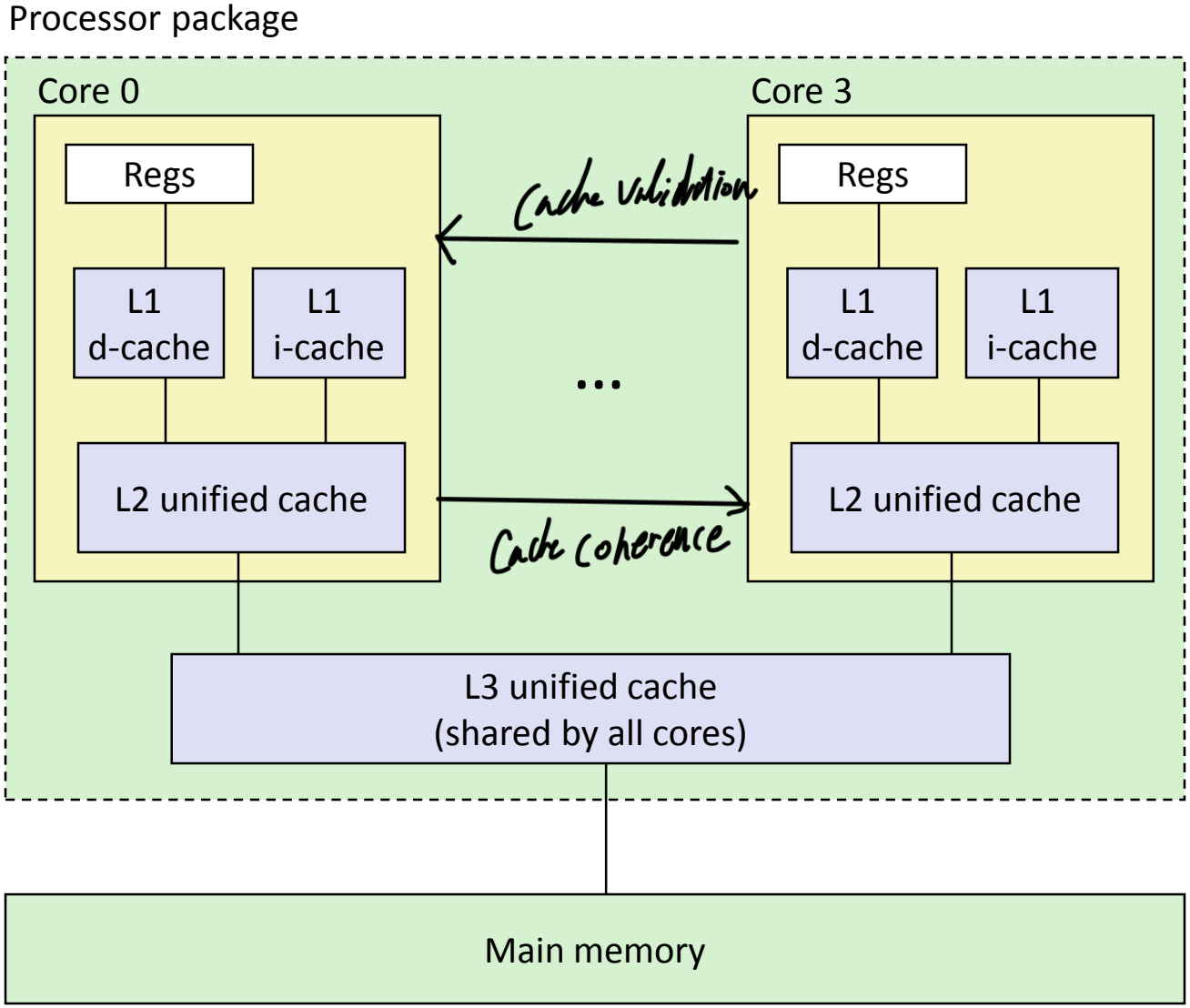
- **Question 1: Where can a Block be Placed?**
  One place (direct-mapped), a few places (set associative), or any place (fully associative)

- **Question 2: How is a Block Found?**
  Indexing (direct-mapped), limited search (set associative), full search (fully associative)

- **Question 3: Which Block is Replaced on a Miss?**
  Typically LRU or random

- **Question 4: How are Writes Handled?**
  Write-through or write-back

Intel Core i7 Cache Hierarchy



L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 11 cycles

L3 unified cache:
8 MB, 16-way,
Access: 30-40 cycles

Block size: 64 bytes for all caches.

Cache Performance Metrics

1 + L1 Latency X percentage + L2 Latency X percentage

- Miss Rate
  - Fraction of memory references not found in cache (misses / accesses)
    = 1 – hit rate
  - Typical numbers (in percentages):
    - 3-10% for L1
    - can be quite small (e.g., < 1%) for L2, depending on size, etc.

$$1 + \sum_{i=1}^{\text{cache depth}} L[i]\text{(latency)} \times P[i]\text{(percentage)}$$

- Hit Time
  - Time to deliver a line in the cache to the processor
    - includes time to determine whether the line is in the cache
  - Typical numbers:
    - 1-2 clock cycle for L1
    - 5-20 clock cycles for L2
- Miss Penalty
  - Additional time required because of a miss
    - typically 50-200 cycles for main memory (Trend: increasing!)

Lets think about those numbers

- Huge difference between a hit and a miss
  - Could be 100x, if just L1 and main memory

- Would you believe 99% hits is twice as good as 97%?
  - Consider:
    cache hit time of 1 cycle
    miss penalty of 100 cycles

    $\frac{0.97 \cdot 1 + 0.03 * 100}{\text{almost } 1}$

  - Average access time:

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

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

- **This is why "miss rate" is used instead of "hit rate"** (miss rate is dominant parameter for access time)

  almost 1 + miss rate * miss penalty

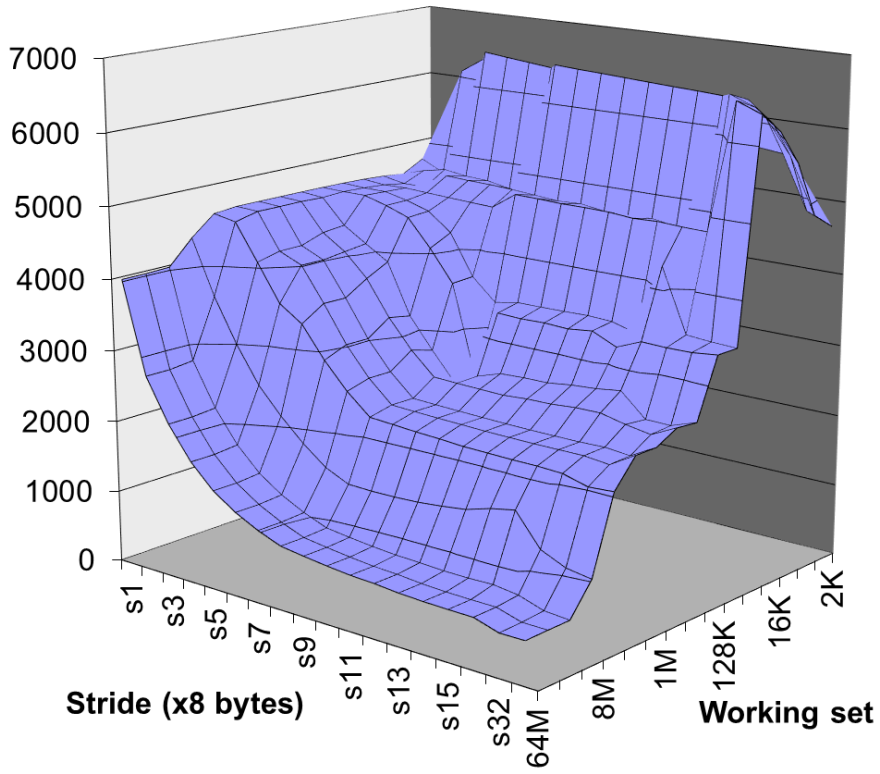
Writing Cache Friendly Code

- 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)

Key point: Our qualitative notion of locality is quantified through our understanding of cache memories.

L1 for hit time optimization : direct-mapped
L2 for miss rate optimization : many as full



Optimizing for the Memory Hierarchy

The Memory Mountain

- **Read throughput** (read bandwidth)
  - Number of bytes read from memory per second (MB/s)

- **Memory mountain**:
  Measured read throughput as a function of spatial and temporal locality.
  - Compact way to characterize memory system performance.

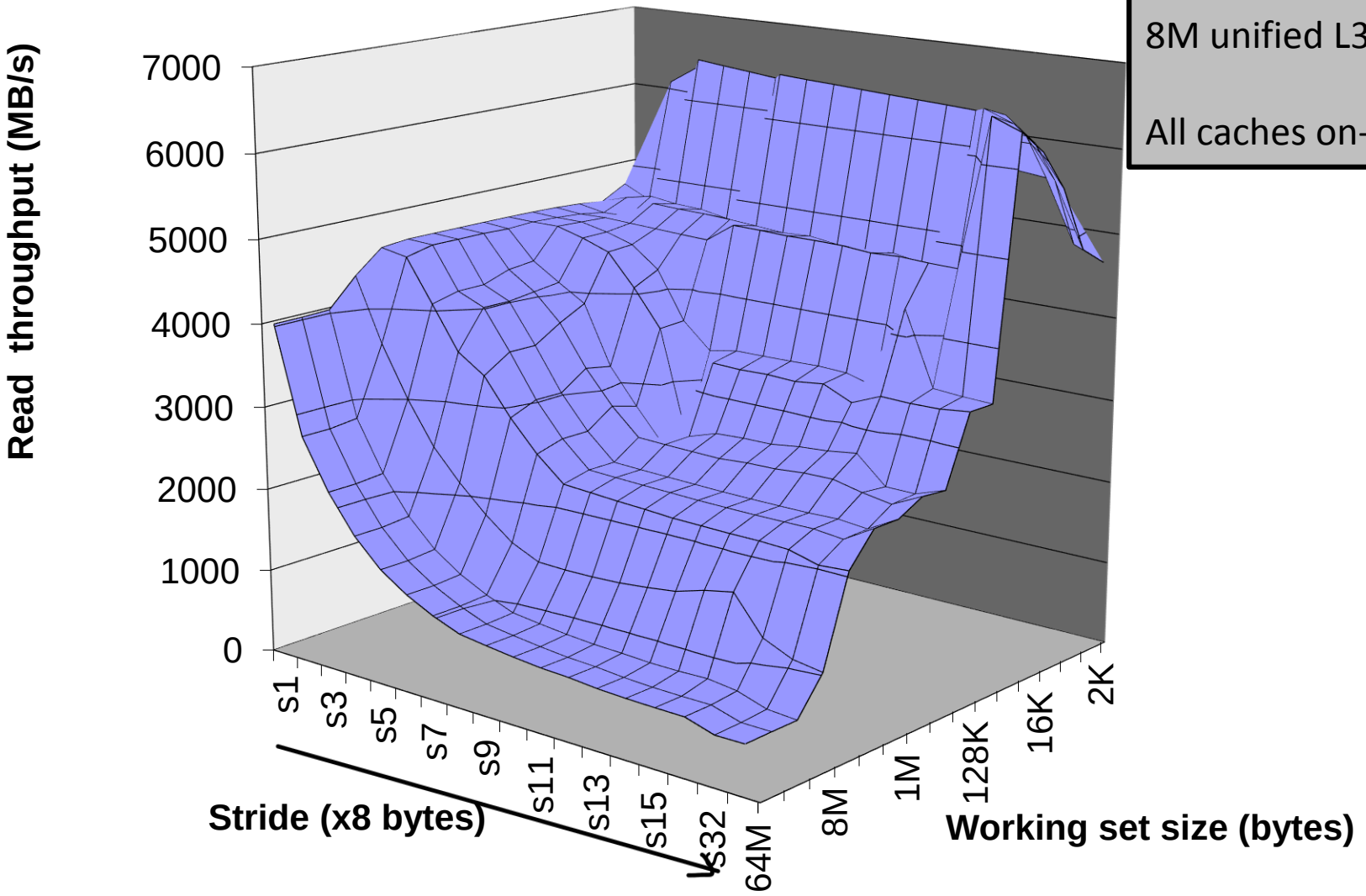
Memory Mountain Test Function

```
/* The test function */
void test(int elems, int stride) {
    int i, result = 0;
    volatile int sink;

    for (i = 0; i < elems; i += stride)
        result += data[i];
    sink = result; /* So compiler doesn't optimize away the loop */
}

/* Run test(elems, stride) and return read throughput (MB/s) */
double run(int size, int stride, double Mhz)
{
    double cycles;
    int elems = size / sizeof(int);

    test(elems, stride);                    /* warm up the cache */
    cycles = fcyc2(test, elems, stride, 0);  /* call test(elems,stride) */
    return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}
```

The Memory Mountain

Intel Core i7
32 KB L1 i-cache
32 KB L1 d-cache
256 KB unified L2 cache
8M unified L3 cache

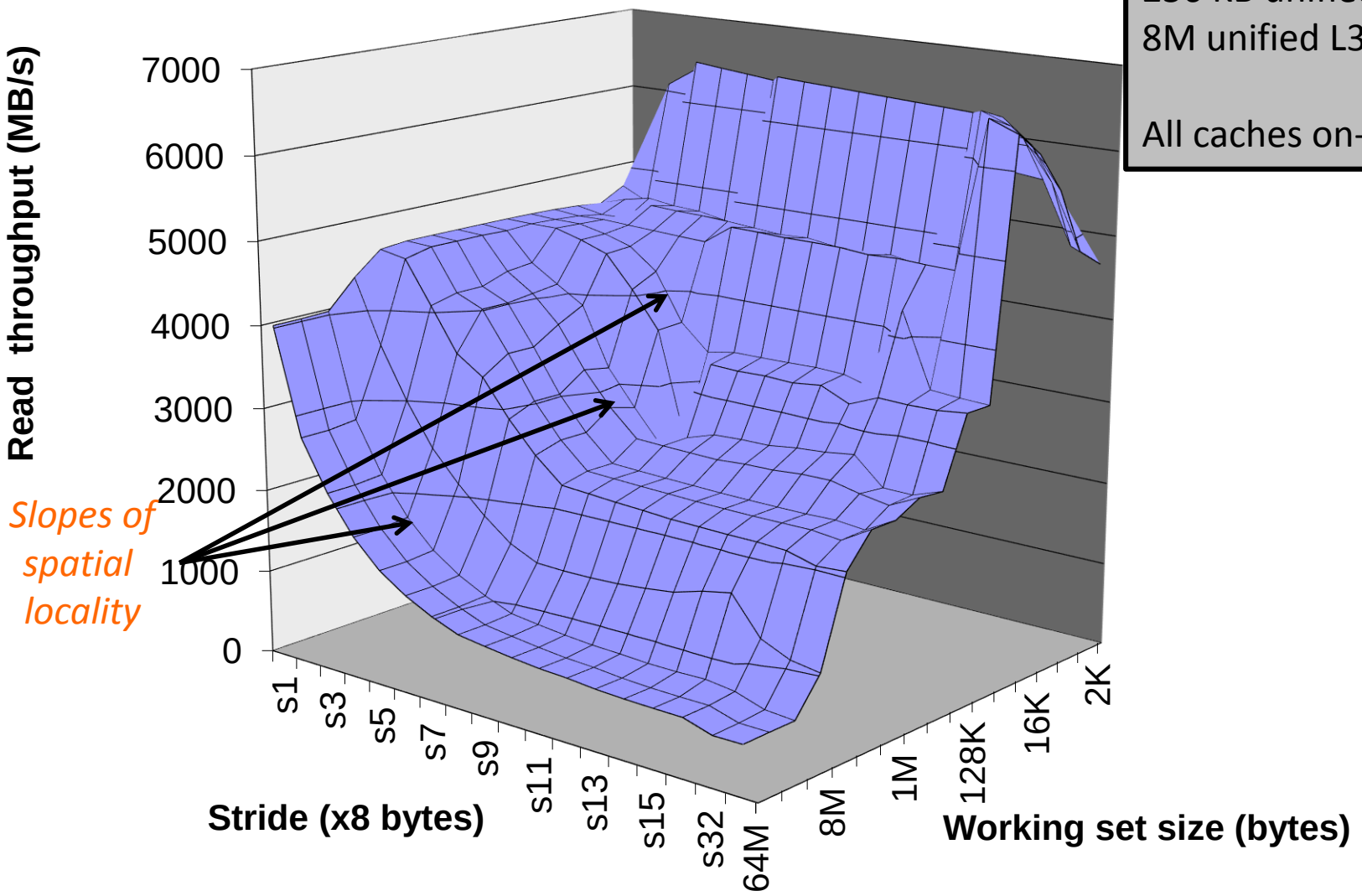
All caches on-chip



The Memory Mountain

Intel Core i7
32 KB L1 i-cache
32 KB L1 d-cache
256 KB unified L2 cache
8M unified L3 cache

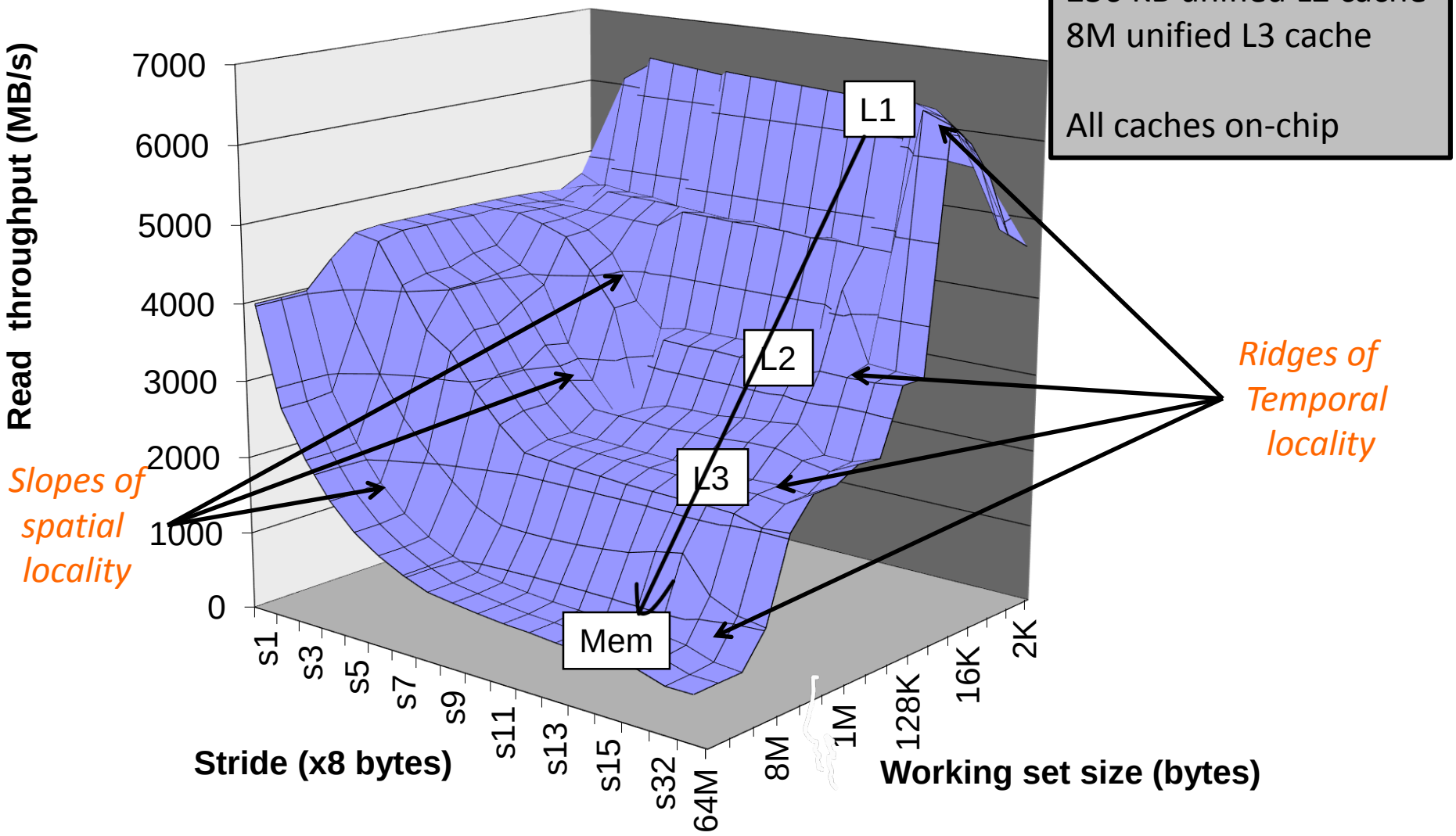
All caches on-chip



The Memory Mountain

Intel Core i7
32 KB L1 i-cache
32 KB L1 d-cache
256 KB unified L2 cache
8M unified L3 cache

All caches on-chip



Qualitative Estimates of Locality

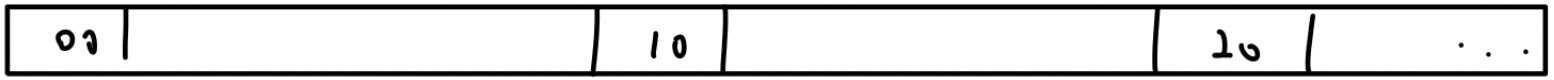
- **Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer!**

- Question: Does this function have good locality with respect to array a?

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Qualitative Estimates of Locality

- Question: Does this function have good locality with respect to array a?




```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Cache miss every time

Qualitative Estimates of Locality

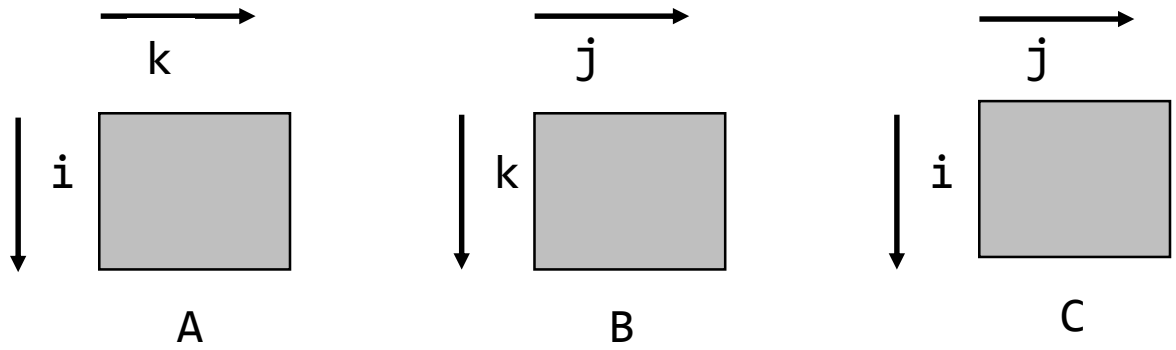
- Question: Can we optimize this loop to maximize locality?

```
int sum_array_3d(int a[N][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                sum += a[k][i][j];
    return sum;
}
```

Miss Rate Analysis for Matrix Multiply

- **Assumptions**
  - Matrix data type: double (64-bit)
  - Line size = 32B (four 64-bit words)
  - 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 innermost loop



Matrix Multiplication Example

- **Description**
  - Multiply two N x N matrices
  - $O(N^3)$ total operations
  - N reads per source element
  - N values summed per destination
    - held in register

Variable sum held in register

```
/* loop order: i j k */
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;
  }
}
```

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(data type), exploit spatial locality
    - compulsory miss rate = sizeof(data type) / B

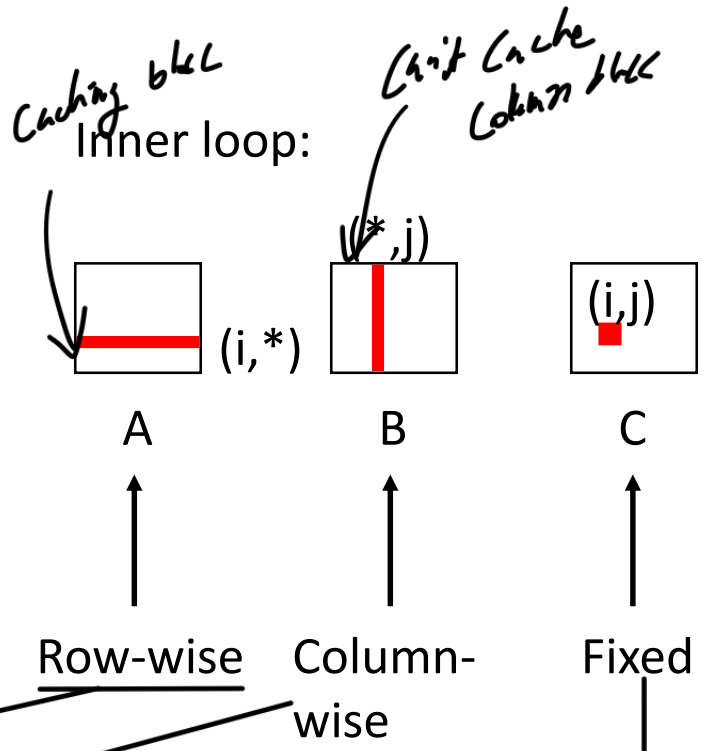
- Stepping through rows in one column:

```
for (i=0; i<n; i++) sum += a[i][0];
```

  - accesses distant elements
  - no spatial locality!
    - compulsory miss rate = 1 (i.e. 100%)

Matrix Multiplication – i-j-k

```
/* loop order: i j k */
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;
  }
}
```



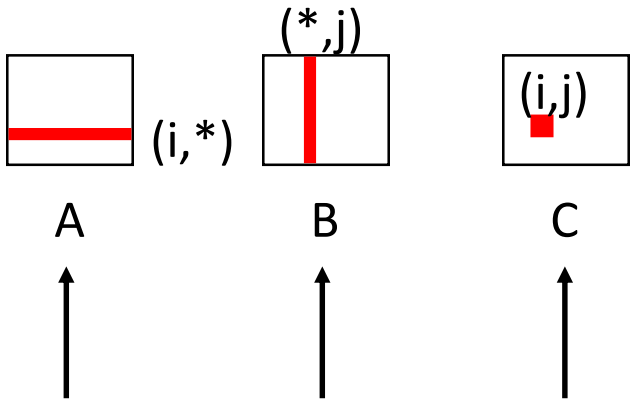
Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication – j-i-k

```
/* loop order: j i k */
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;
  }
}
```

Inner loop:



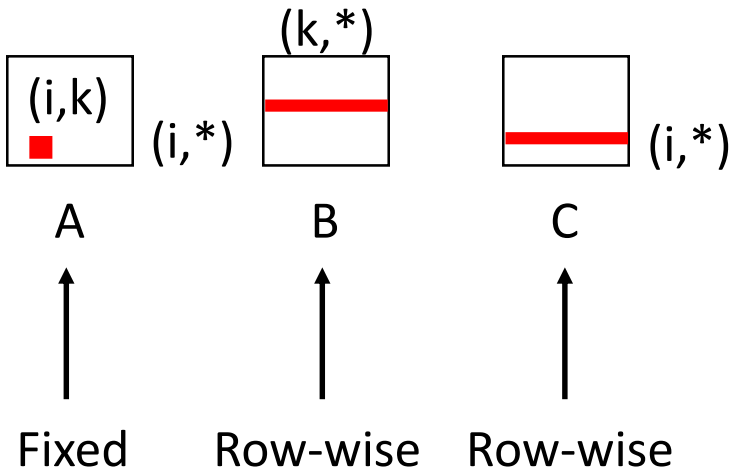
Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication – k-i-j

```
/* loop order: k i j */
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:



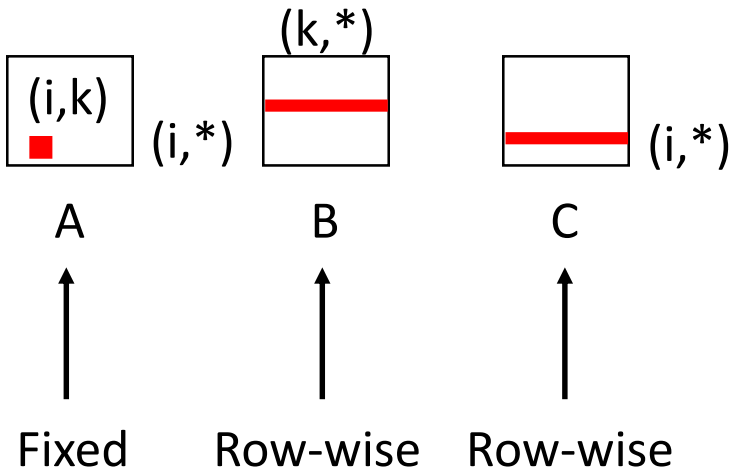
Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication – i-k-j

```
/* loop order: i k j */
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:



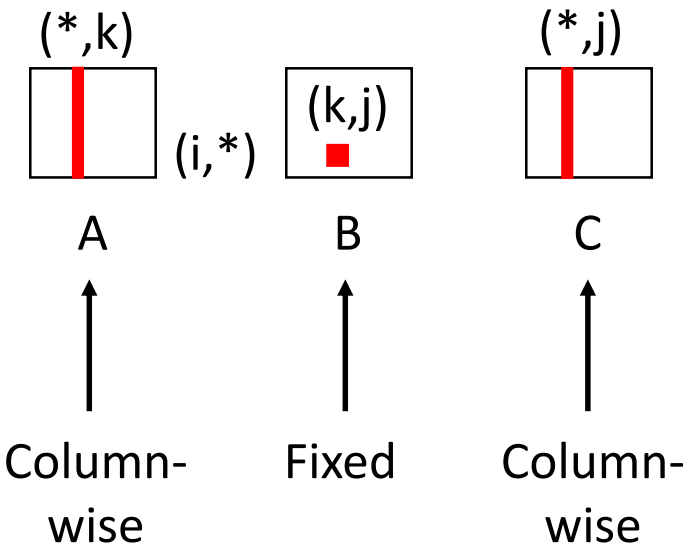
Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication – j-k-i

```
/* loop order: j k i */
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:



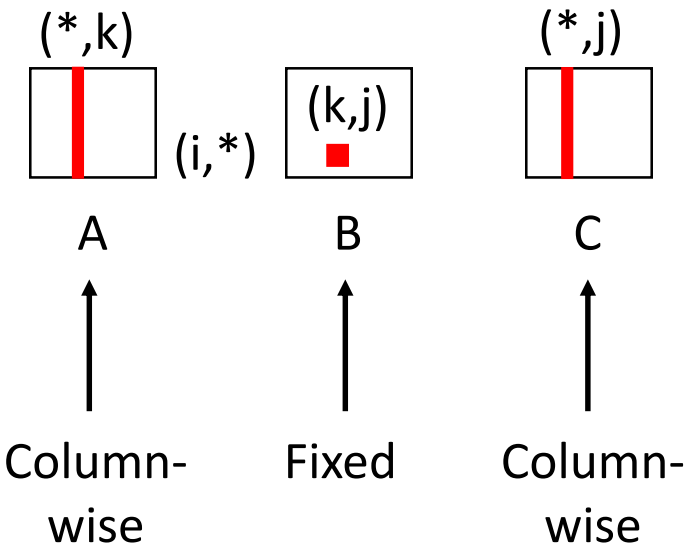
Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Matrix Multiplication – k-j-i

```
/* loop order: k j i */
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 = 1.25

```
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):

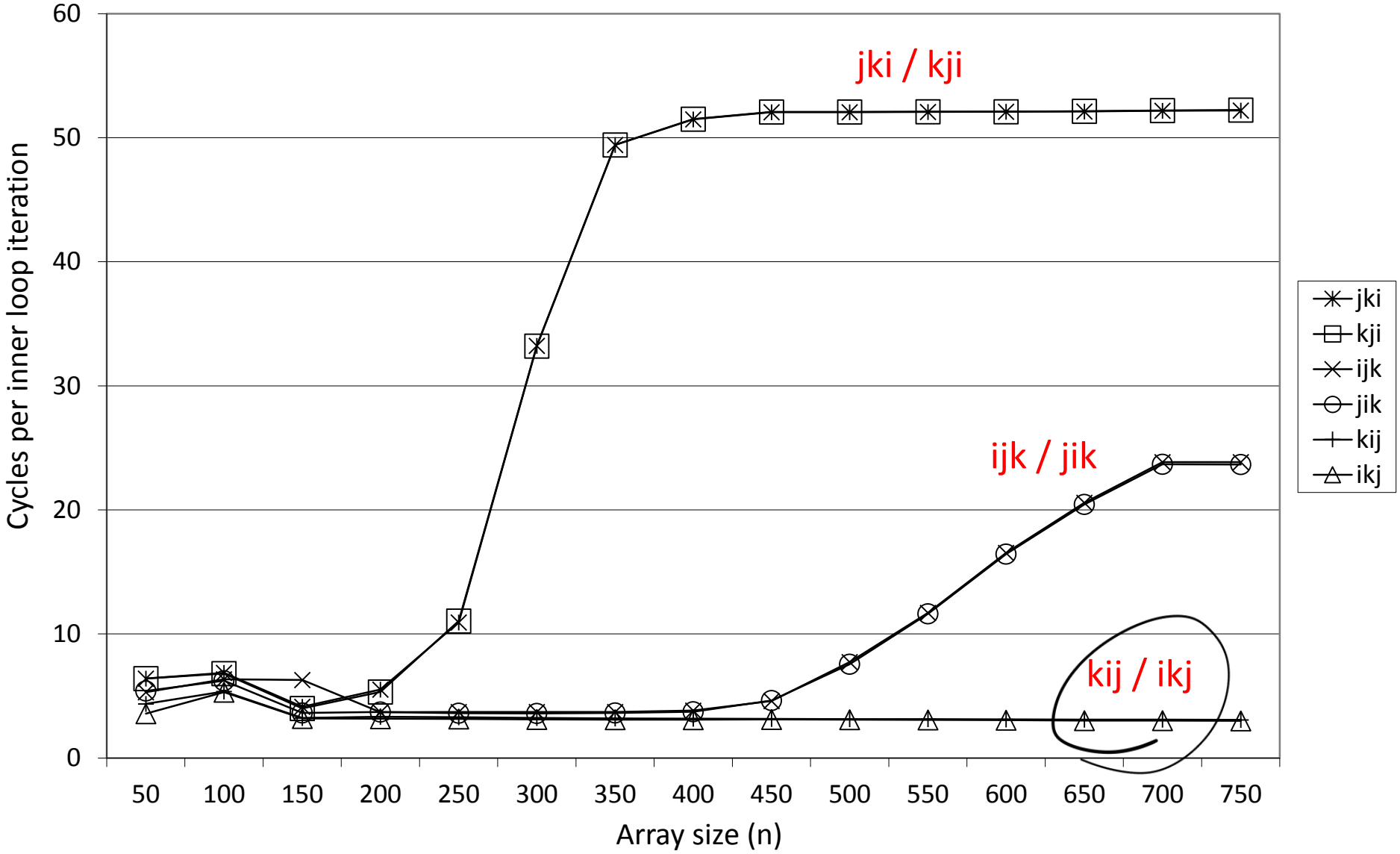
- 2 loads, 1 store
- misses/iter = 0.5

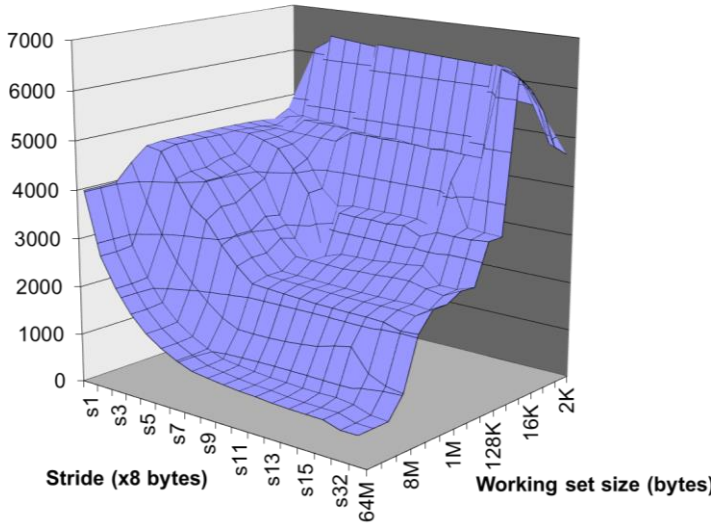
```
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

jki (& kji):

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

Core i7 Matrix Multiply Performance





Module Summary

Summary

- Memory hierarchies are an optimization resulting from a perfect match between memory technology and two types of program locality
  - Temporal locality
  - Spatial locality
- The goal is to provide a “virtual” memory technology (an illusion) that has an access time of the highest-level memory with the size and cost of the lowest-level memory
- Cache memory is an instance of a memory hierarchy
  - exploits both temporal and spatial localities
  - direct-mapped caches are simple and fast but have higher miss rates
  - set-associative caches have lower miss rates but are complex and slow
  - multilevel caches are becoming increasingly popular
- Programmer can optimize for cache performance
  - How data structure are organized
  - How data are accessed (nested loop structure)

LAB