Memory Hierarchy

Hung-Wei Tseng

Disney+

Disney · PIXAR

INSIDE OUT

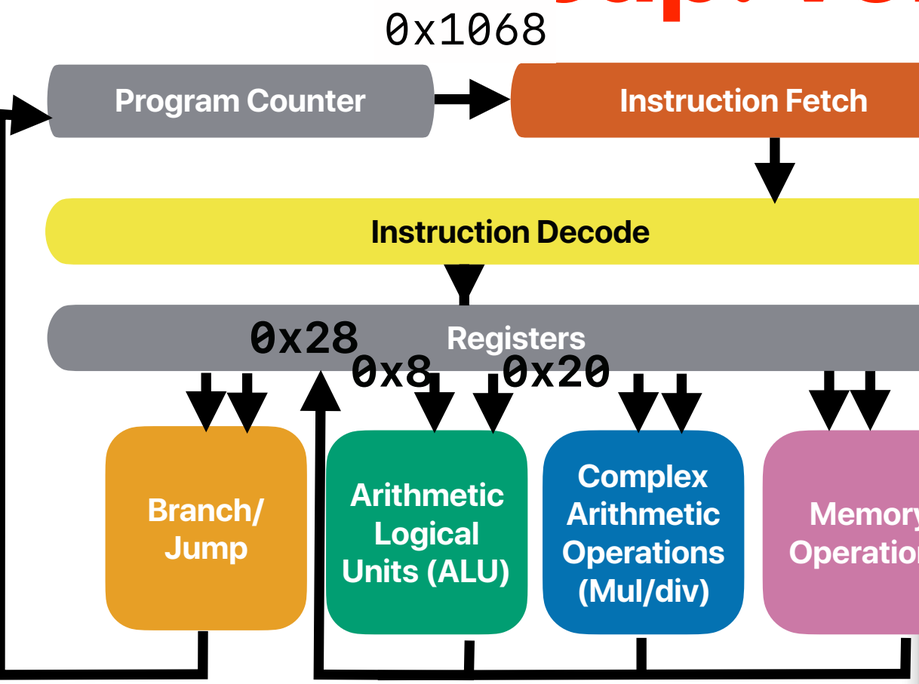
GET DISNEY+

TRAILER

PG 2015 • 1h 35m • Coming of age, Family, Animation

When 11-year-old Riley moves to a new city, her Emotions team up to help her through the transition. Joy, Fear, Anger, Disgust and Sadness work together, but when Joy and Sadness get lost, they must journey through unfamiliar places to get back home.

Recap: von Neumann architecture

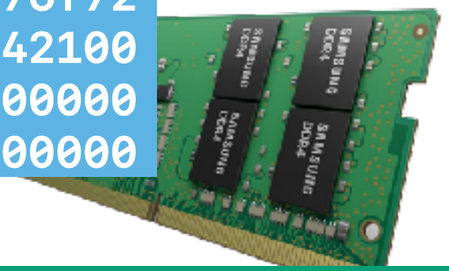


Processor

4883ec08

sub $0x8,%rsp

Instructions	Data
f30f1efa	08400000
4883ec08	00000100
488d3d95	02004865
0f0000e8	6c6c6f2c
dcffffff	20776f72
31c04883	6c642100
c408c30f	00000000
1f440000	00000000



Memory

```
int main(){
    pr
}
```

Instructions	Data
f30f1efa	08400000
4883ec08	00000100
488d3d95	02004865
0f0000e8	6c6c6f2c
dcffffff	20776f72
31c04883	6c642100
c408c30f	00000000
1f440000	00000000

Storage

By loading different programs into memory, your computer can perform different functions

Recap: Speedup and Amdahl's Law?

- Definition of "Speedup of Y over X" or say Y is n times faster than X: $speedup_{Y\_over\_X} = n = \frac{Execution\ Time_X}{Execution\ Time_Y}$
- Amdahl's Law — $Speedup_{enhanced}(f, s) = \frac{1}{(1-f) + \frac{f}{s}}$ $Speedup_{max}(f, \infty) = \frac{1}{(1-f)}$
  - Corollary 1 — each optimization has an upper bound
  - Corollary 2 — make the common case (the most time consuming case) fast!
  - Corollary 3: Optimization has a moving target

$Speedup_{max}(f_1, \infty) = \frac{1}{(1-f_1)}$
$Speedup_{max}(f_2, \infty) = \frac{1}{(1-f_2)}$
$Speedup_{max}(f_3, \infty) = \frac{1}{(1-f_3)}$
$Speedup_{max}(f_4, \infty) = \frac{1}{(1-f_4)}$

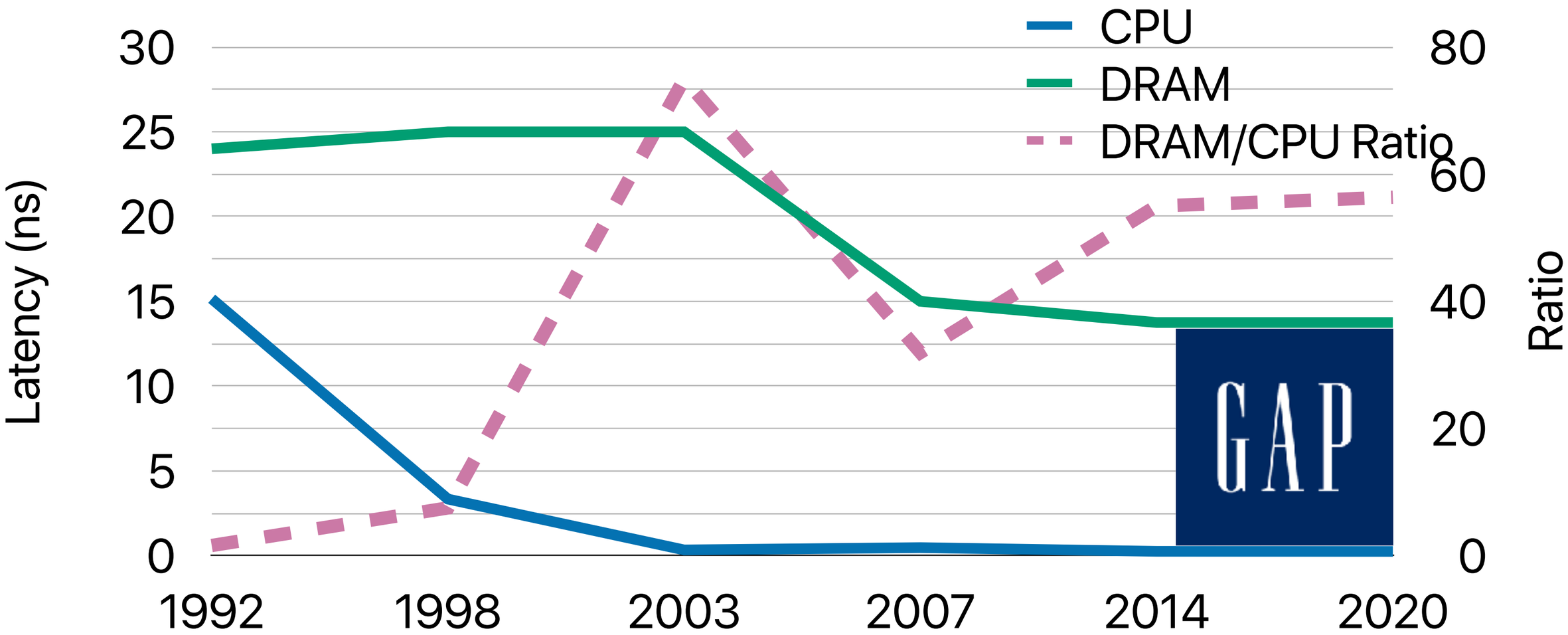
  - Corollary 4: Exploiting more parallelism from a program is the key to performance gain in modern architectures $Speedup_{parallel}(f_{parallelizable}, \infty) = \frac{1}{(1-f_{parallelizable})}$
  - Corollary 5: Single-core performance still matters

$Speedup_{parallel}(f_{parallelizable}, \infty) = \frac{1}{(1-f_{parallelizable})}$

Outline

- The memory-wall problem
- The “predictable” code behavior
- Designing a cache that captures the predictability

The “latency” gap between CPU and DRAM



CPU	i486	Pentium II	Pentium 4	Core 2	Core i7-4790K	Core i5-10600K
DRAM	SDRAM	DDR	DDR2	DDR3	DDR4	DDR5

20% is under-estimating …

Instruction class	MIPS examples	HLL correspondence	Frequency	
			Integer	Ft. pt.
Arithmetic	add, sub, addi	Operations in assignment statements	16%	48%
Data transfer	lw, sw, lb, lbu, lh, lhu, sb, lui	References to data structures, such as arrays	35%	36%
Logical	and, or, nor, andi, ori, sll, srl	Operations in assignment statements	12%	4%
Conditional branch	beq, bne, slt, slti, sltiu	*If* statements and loops	34%	8%
Jump	j, jr, jal	Procedure calls, r eturns, and *case/switch* statements	2%	0%

FIGURE 2.48 MIPS instruction classes, examples, correspondence to high-level program language constructs, and percentage of MIPS instructions executed by category for the average integer and floating point SPEC CPU2006 benchmarks.

Figure 3.24 in Chapter 3 shows average percentage of the individual MIPS instructions executed.

Recap: Speedup and Amdahl's Law?

- Definition of "Speedup of Y over X" or say Y is n times faster than X: $speedup_{Y\_over\_X} = n = \frac{Execution\ Time_X}{Execution\ Time_Y}$
- Amdahl's Law — $Speedup_{enhanced}(f, s) = \frac{1}{(1-f) + \frac{f}{s}}$ $Speedup_{max}(f, \infty) = \frac{1}{(1-f)}$
  - Corollary 1 — each optimization has an upper bound
  - **Corollary 2 — make the common case (the most time consuming case) fast!**
  - Corollary 3: Optimization has a moving target
  - Corollary 4: Exploiting more parallelism from a program is the key to performance gain in modern architectures $Speedup_{parallel}(f_{parallelizable}, \infty) = \frac{1}{(1-f_{parallelizable})}$
  - Corollary 5: Single-core performance still matters

$Speedup_{max}(f_1, \infty) = \frac{1}{(1-f_1)}$
$Speedup_{max}(f_2, \infty) = \frac{1}{(1-f_2)}$
$Speedup_{max}(f_3, \infty) = \frac{1}{(1-f_3)}$
$Speedup_{max}(f_4, \infty) = \frac{1}{(1-f_4)}$

$Speedup_{parallel}(f_{parallelizable}, \infty) = \frac{1}{(1-f_{parallelizable})}$

Take-aways: inside out our memory hierarchy

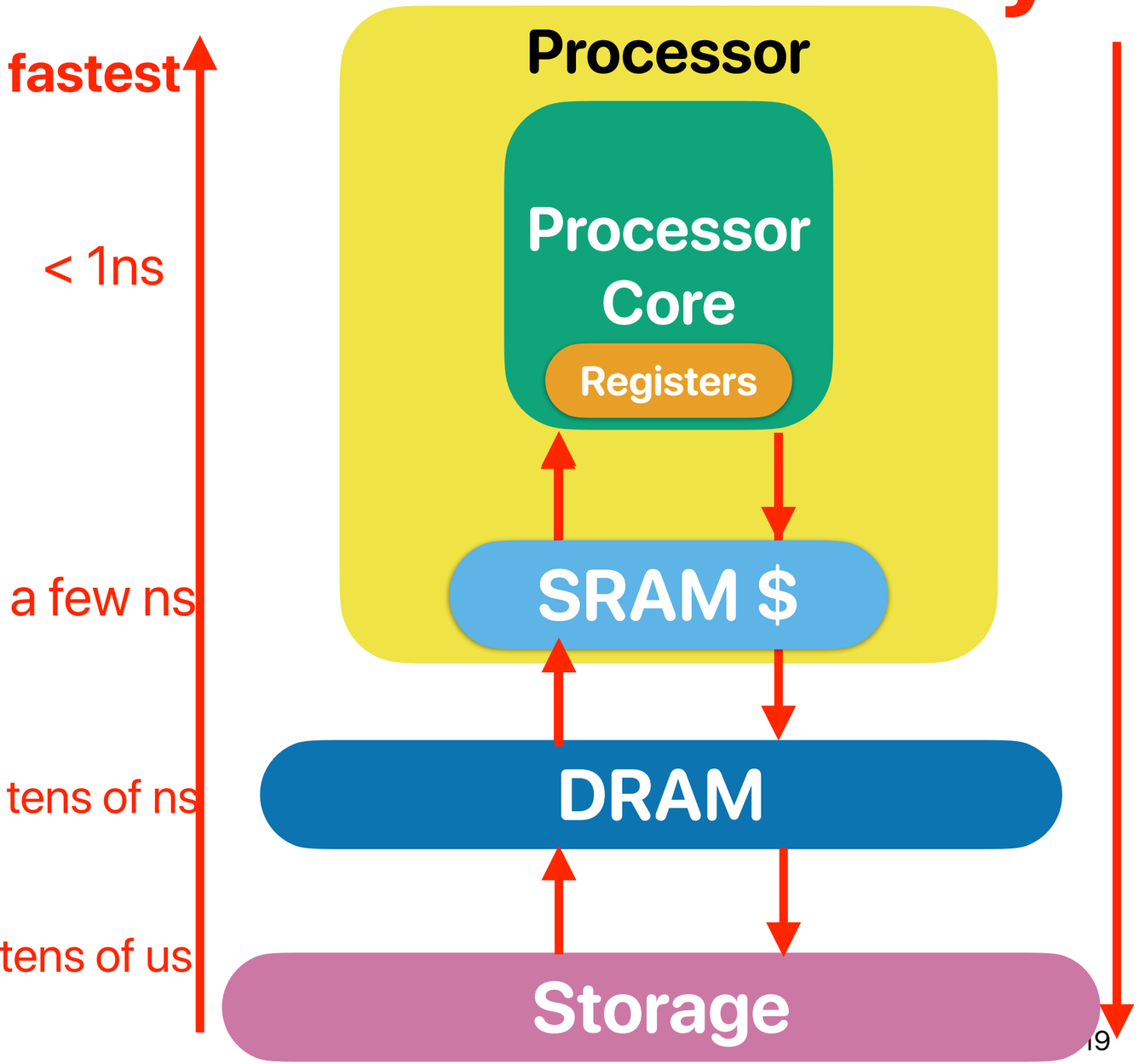
- Memory access time is the most critical performance problem
  - One memory operation is as expensive as 50 arithmetic operations
  - Processor has to fetch instructions from memory
  - We have an average of 33% of data memory access instructions!

Alternatives?

Memory technology	Typical access time	$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$10–$20
Flash semiconductor memory	5,000–50,000 ns	$0.75–$1.00
Magnetic disk	5,000,000–20,000,000 ns	$0.05–$0.10

Fast, but expensive $$$

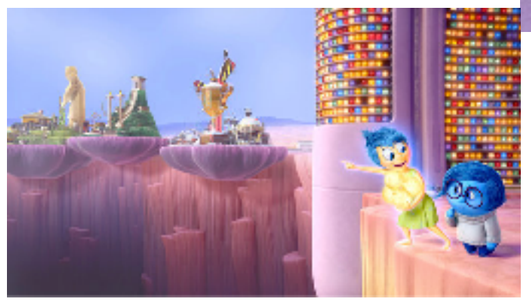
Memory Hierarchy



32 or 64 words

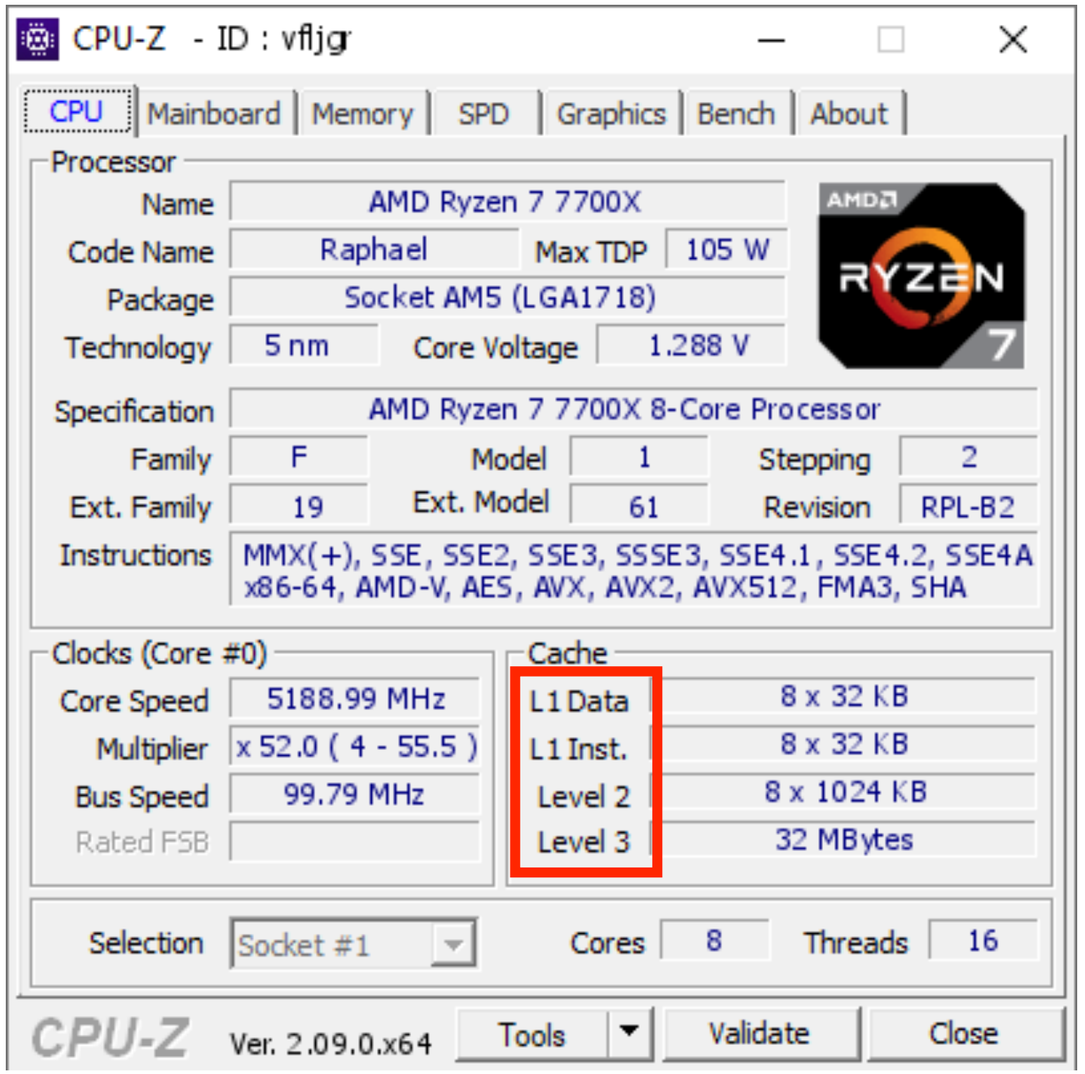
KBs ~ MBs

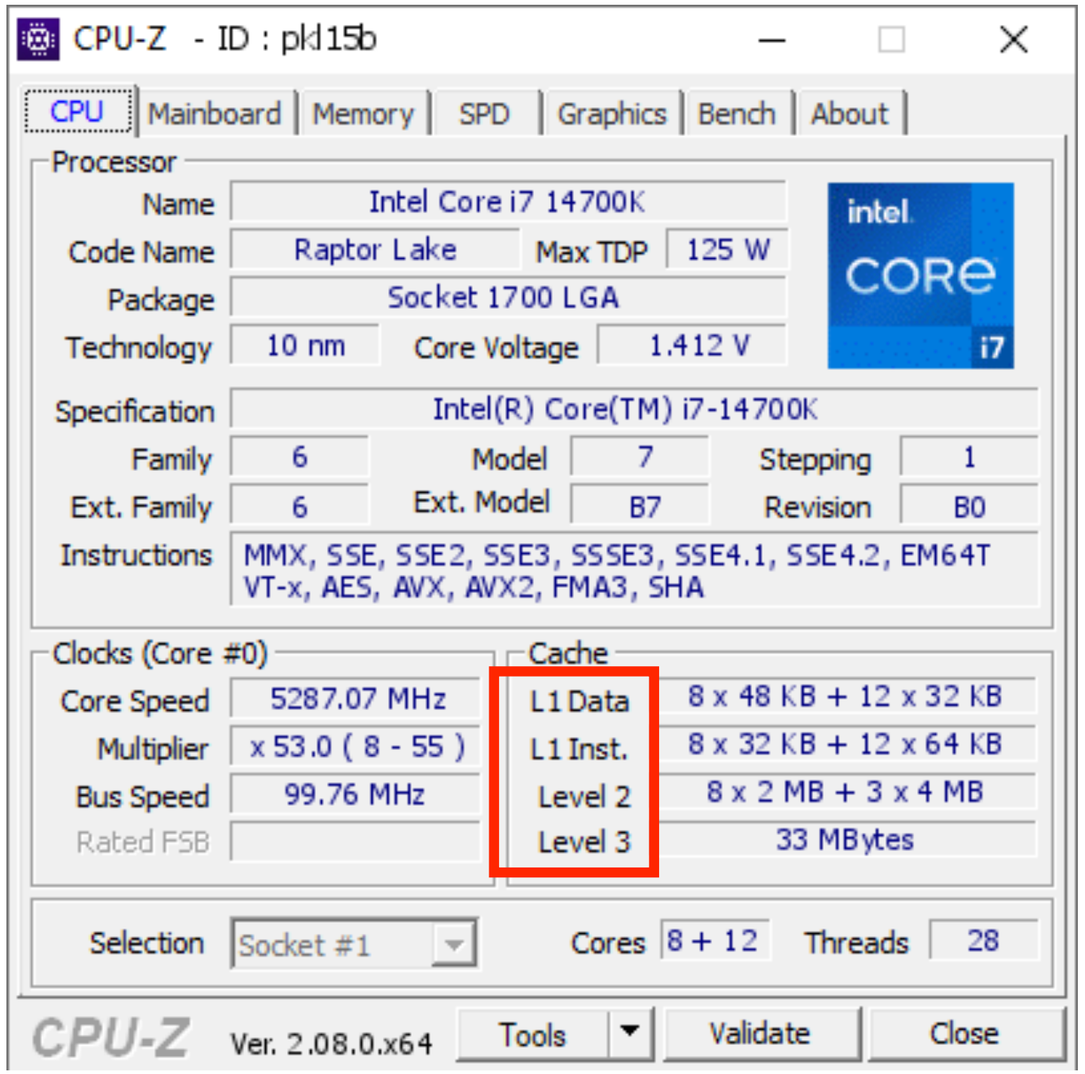
GBs



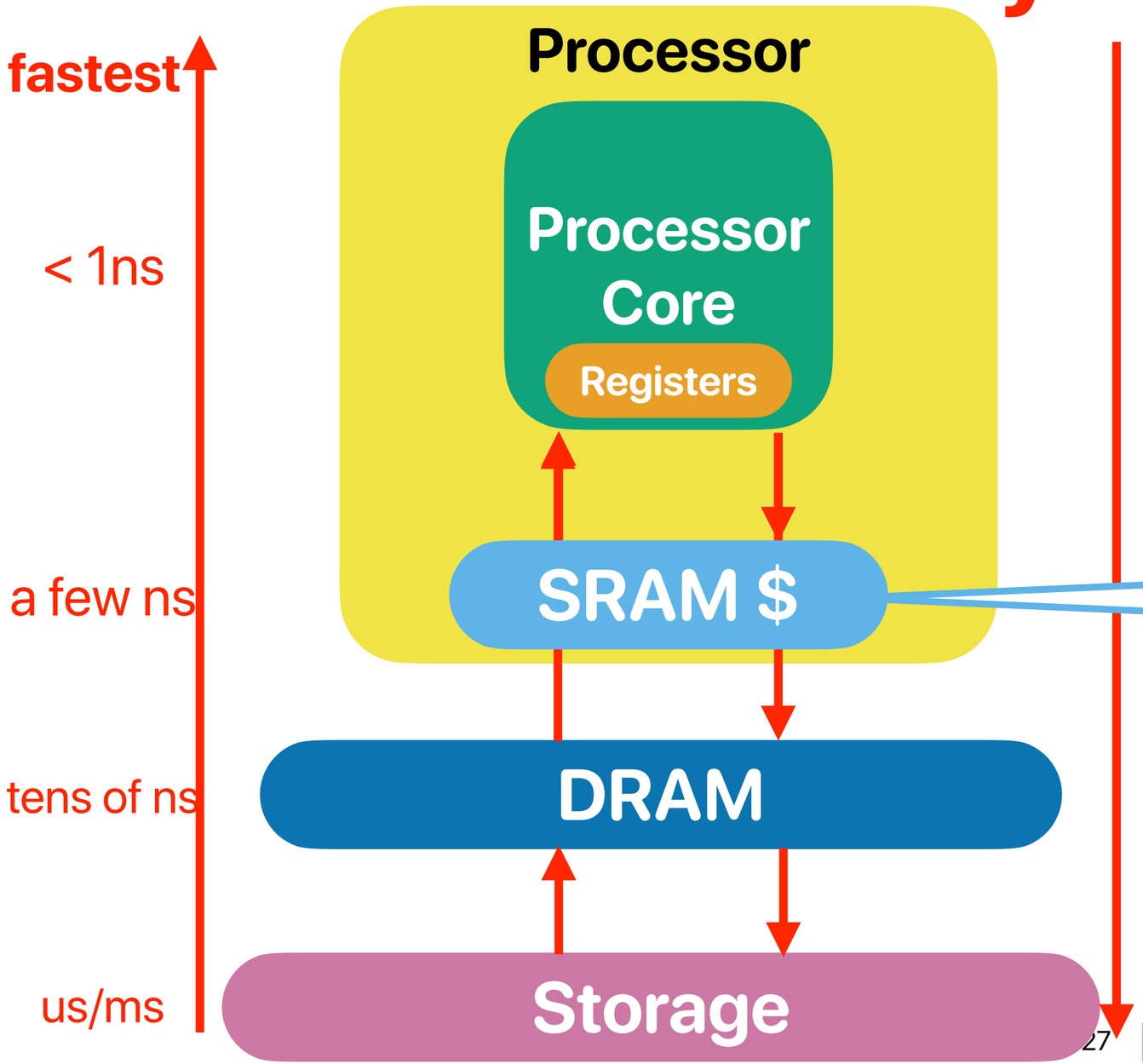
TBs

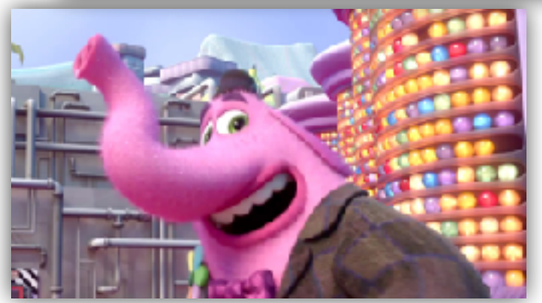
L1? L2? L3?


CPU-Z - ID : vfljg
CPU
Mainboard
Memory
SPD
Graphics
Bench
About
Processor
Name
AMD Ryzen 7 7700X
Code Name
Raphael
Max TDP
105 W
Package
Socket AM5 (LGA1718)
Technology
5 nm
Core Voltage
1.288 V
Specification
AMD Ryzen 7 7700X 8-Core Processor
Family
F
Model
1
Stepping
2
Ext. Family
19
Ext. Model
61
Revision
RPL-B2
Instructions
MMX(+), SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, SSE4A
x86-64, AMD-V, AES, AVX, AVX2, AVX512, FMA3, SHA
Clocks (Core #0)
Core Speed
5188.99 MHz
Multiplier
x 52.0 (4 - 55.5)
Bus Speed
99.79 MHz
Rated FSB
Cache
L1 Data
8 x 32 KB
L1 Inst.
8 x 32 KB
Level 2
8 x 1024 KB
Level 3
32 MBytes
Selection
Socket #1
Cores
8
Threads
16
CPU-Z
Ver. 2.09.0.x64
Tools
Validate
Close

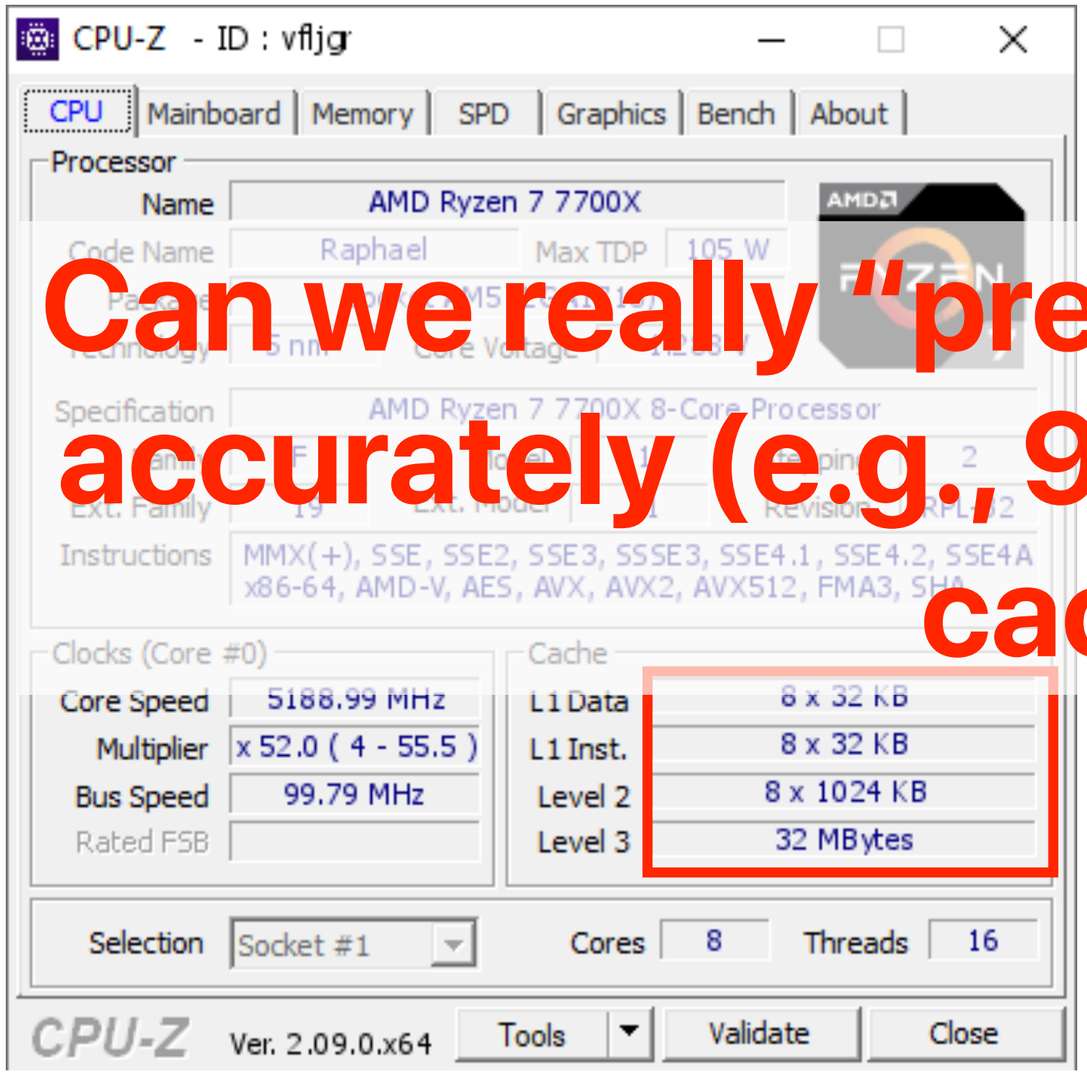

CPU-Z - ID : pkl15b
CPU
Mainboard
Memory
SPD
Graphics
Bench
About
Processor
Name
Intel Core i7 14700K
Code Name
Raptor Lake
Max TDP
125 W
Package
Socket 1700 LGA
Technology
10 nm
Core Voltage
1.412 V
Specification
Intel(R) Core(TM) i7-14700K
Family
6
Model
7
Stepping
1
Ext. Family
6
Ext. Model
B7
Revision
B0
Instructions
MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T
VT-x, AES, AVX, AVX2, FMA3, SHA
Clocks (Core #0)
Core Speed
5287.07 MHz
Multiplier
x 53.0 (8 - 55)
Bus Speed
99.76 MHz
Rated FSB
Cache
L1 Data
8 x 48 KB + 12 x 32 KB
L1 Inst.
8 x 32 KB + 12 x 64 KB
Level 2
8 x 2 MB + 3 x 4 MB
Level 3
33 MBytes
Selection
Socket #1
Cores
8 + 12
Threads
28
CPU-Z
Ver. 2.08.0.x64
Tools
Validate
Close

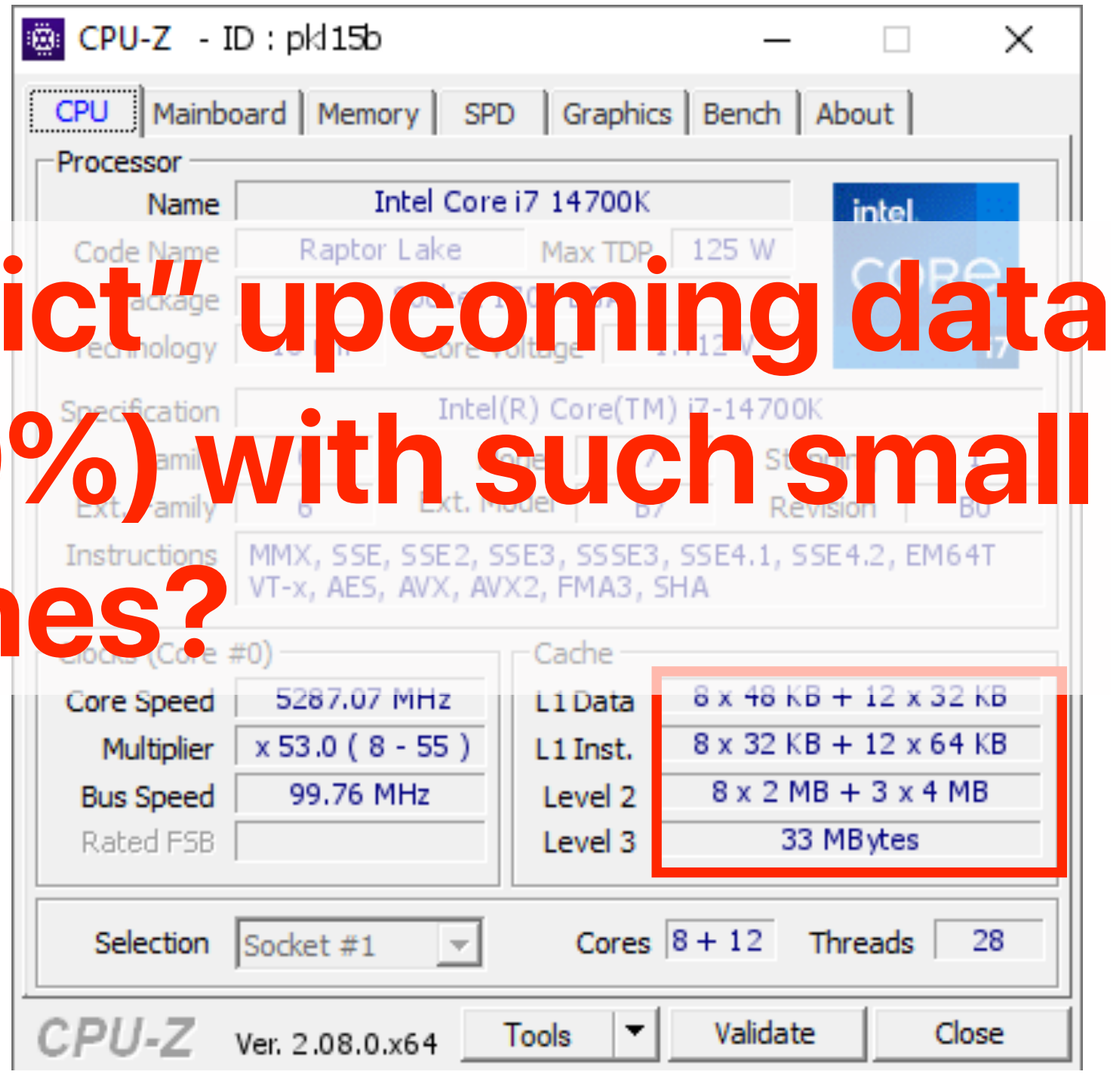
Memory Hierarchy





L1? L2? L3?





Can we really "predict" upcoming data accurately (e.g., 90%) with such small caches?

Take-aways: inside out our memory hierarchy

- Memory access time is the most critical performance problem
  - One memory operation is as expensive as 50 arithmetic operations
  - Processor has to fetch instructions from memory
  - We have an average of 33% of data memory access instructions!
- Hierarchical caching with small amount of SRAMs will work if we can efficiently capture data and instructions

The predictability of your code

The Machine Learning Inference Pipeline

Feature Extraction

Classification

Convolution + ReLU

Pooling

Convolution + ReLU

Pooling

Fully Connected

Output Prediction

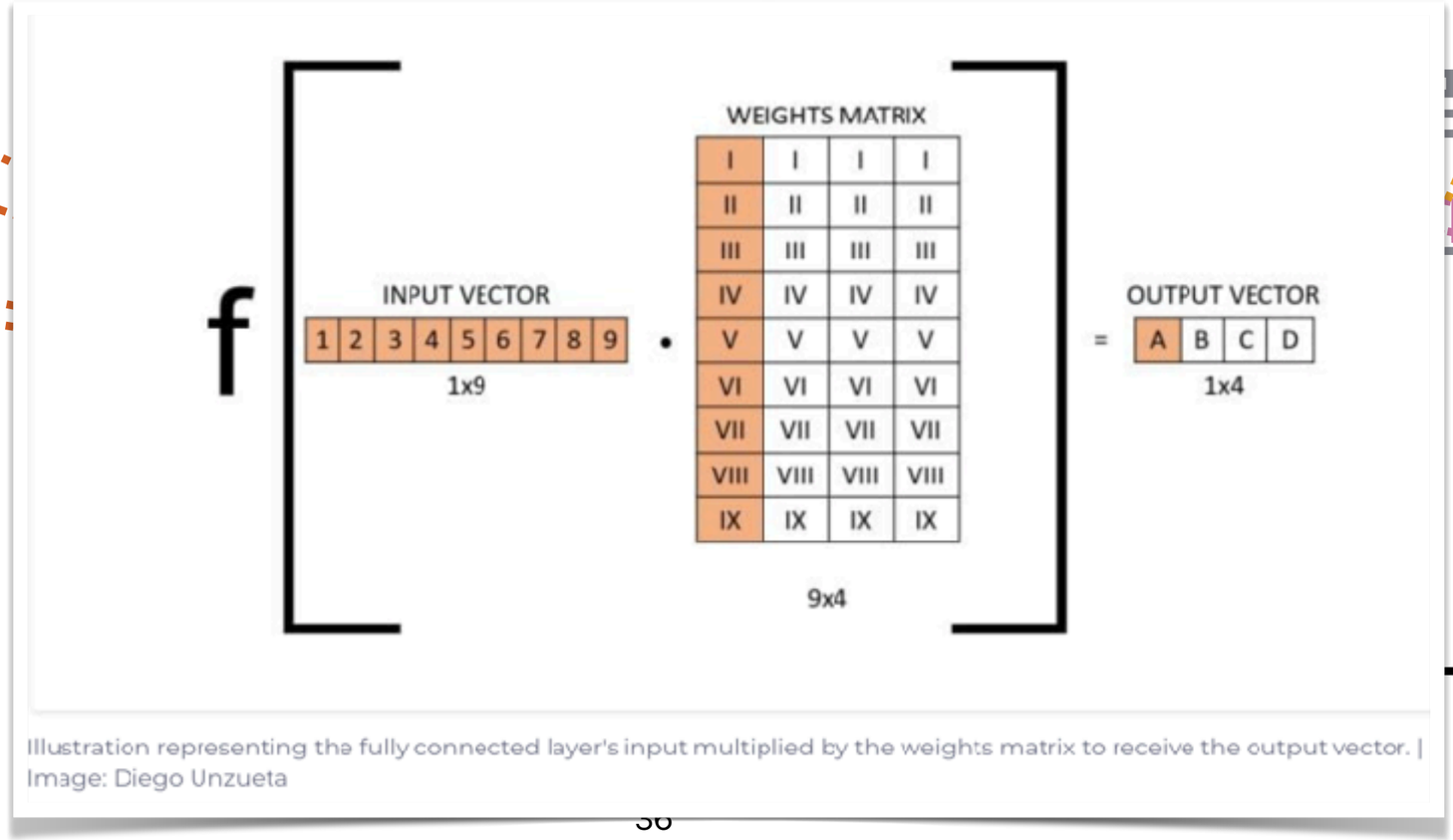
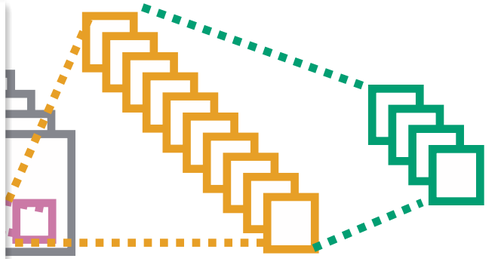


Illustration representing the fully connected layer's input multiplied by the weights matrix to receive the output vector. | Image: Diego Unzueta



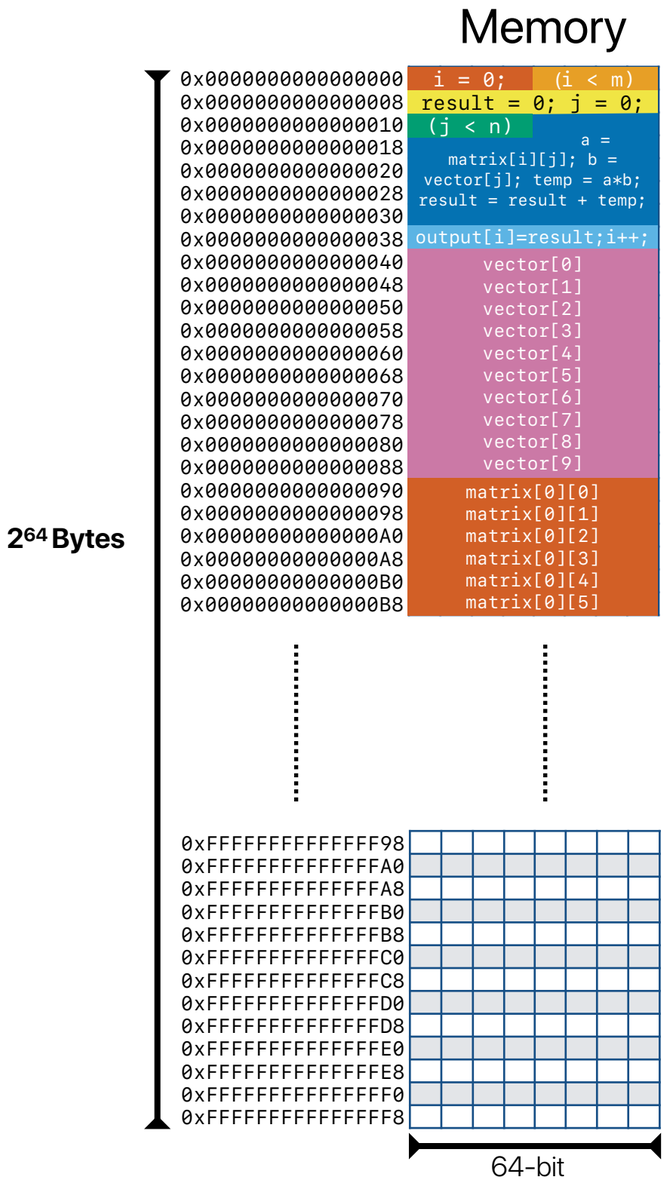
Triton (1)

Cat (0)

Human (0)

Tiger (0)

Data/instructions in the memory



Code also has locality

```
for(uint64_t i = 0; i < m; i++) {
    result = 0;
    for(uint64_t j = 0; j < n; j++) {
        result += matrix[i][j]*vector[j];
    }
    output[i] = result;
}
```

```
i = 0;
while(i < m) {
    result = 0;
    j = 0;
    while(j < n) {
        a = matrix[i][j];
        b = vector[j];
        temp = a*b;
        result = result + temp;
    }
    output[i] = result;
    i++;
}
```

keep going to the next instruction — spatial locality

repeat many times — temporal locality!

Locality

- Spatial locality — application tends to visit nearby stuffs in the memory
  - Code — the current instruction, and then the next instruction
  - Data — the current data, and then the next data
- Temporal locality — application revisit the same thing again and again
  - Code — loops, frequently invoked functions
    - Typically tens of static instructions — at most several KBs
  - Data — program can read/write the same data many times (e.g., vectors in matrix-vector product)

Most of time, your program is just visiting a very small amount of data/instructions within a given window

Take-aways: inside out our memory hierarchy

- Memory access time is the most critical performance problem
  - One memory operation is as expensive as 50 arithmetic operations
  - Processor has to fetch instructions from memory
  - We have an average of 33% of data memory access instructions!
- Hierarchical caching with small amount of SRAMs will work if we can efficiently capture data and instructions
- Caching is possible! Most of time, we only work on a small amount of data!
  - Spatial locality
  - Temporal locality

3:00

How do you tell people where you live in your home town?

Outline

- Designing a cache that captures the predictability
- The A, B, Cs of caches
- Estimating how cache friendly is our code

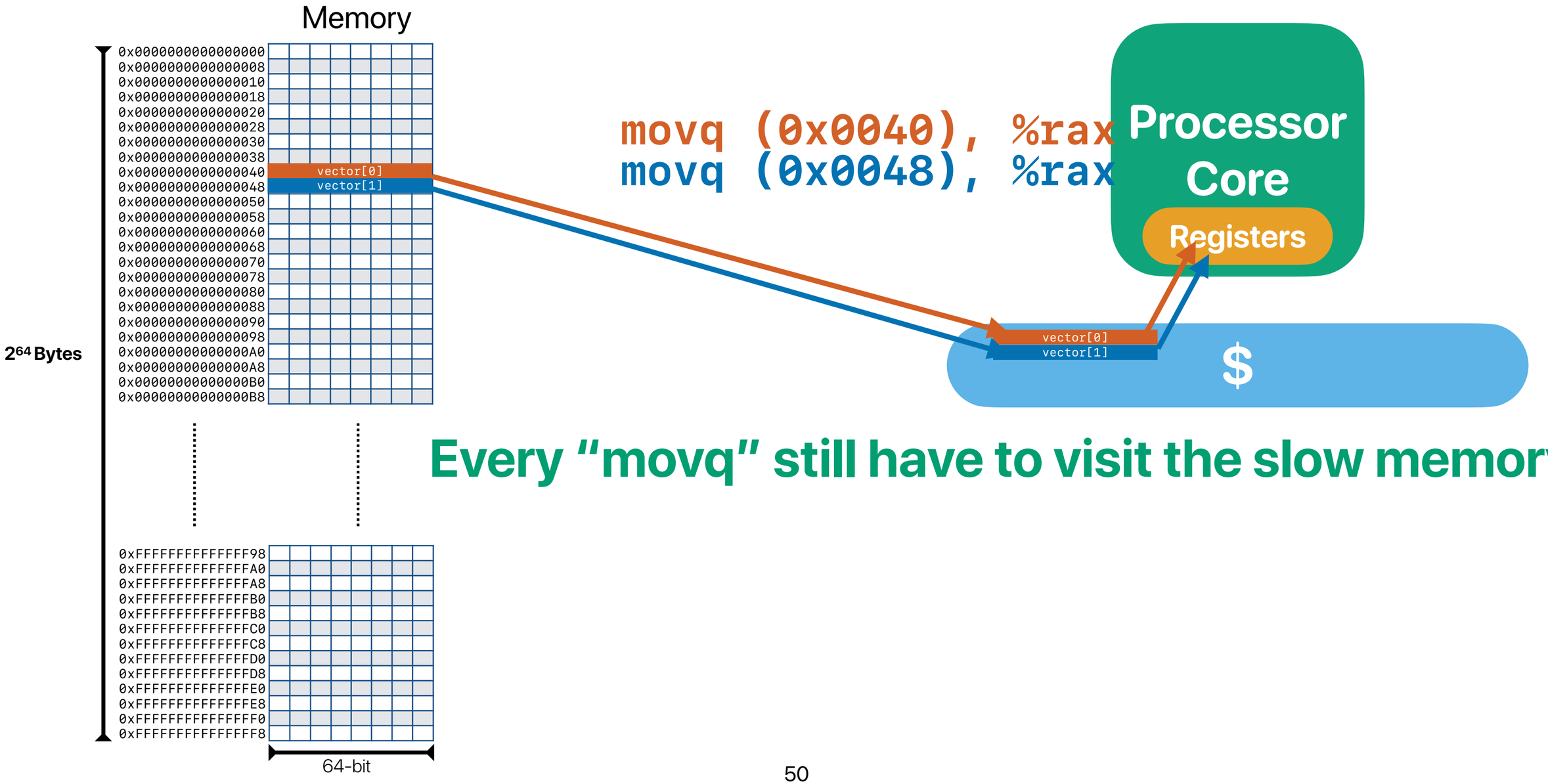
Designing a hardware to exploit locality

- Spatial locality — application tends to visit nearby stuffs in the memory

  We need to "cache consecutive memory locations" every time — the cache should store a "block" of code/data

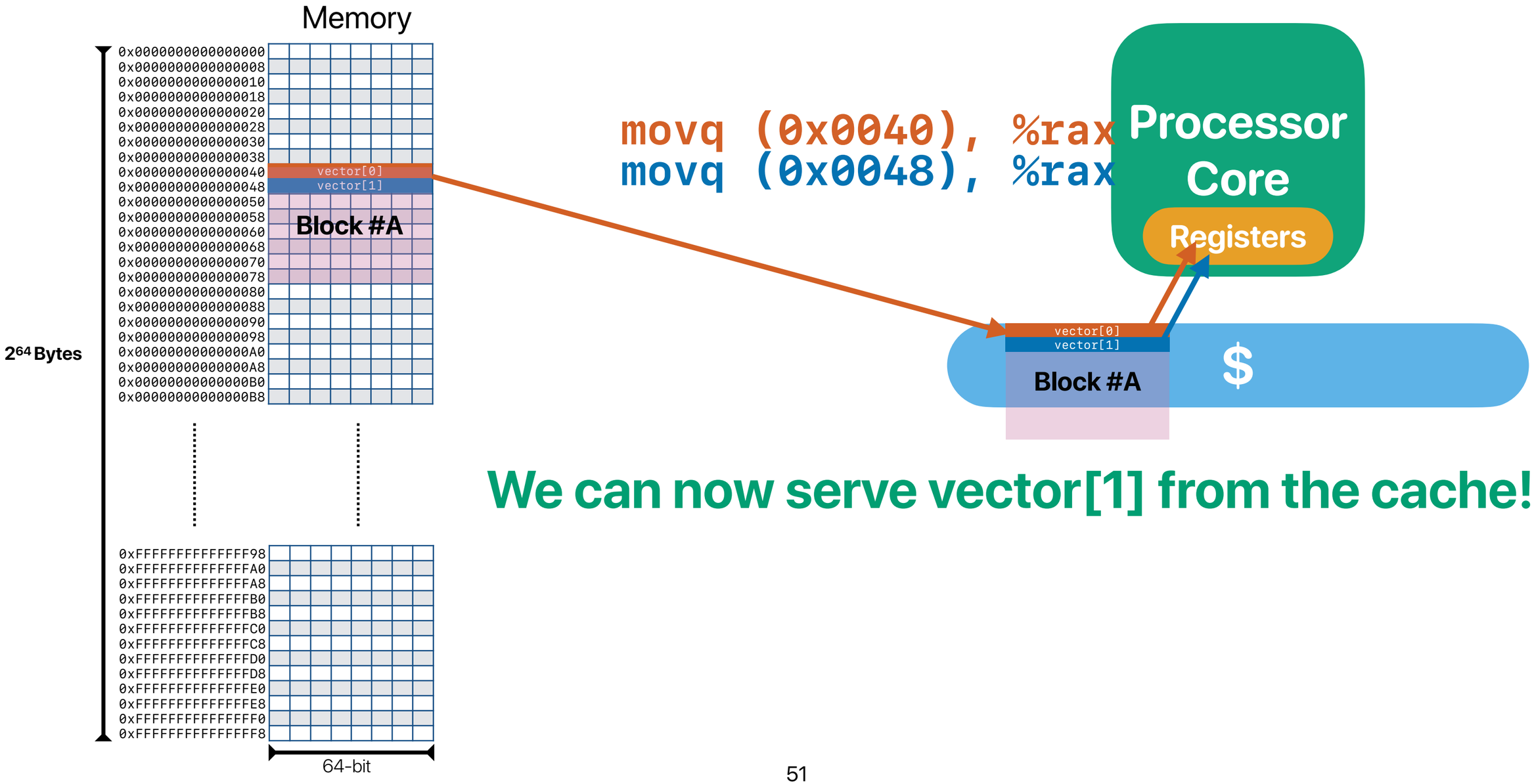
- Temporal locality — application revisit the same thing again and again
  - Code — loops, frequently invoked functions
    - Typically tens of static instructions — at most several KBs
  - Data — program can read/write the same data many times (e.g., vectors in matrix-vector product)

If we just cache what has requested



Every “movq” still have to visit the slow memory!

If we can cache a block of data



We can now serve vector[1] from the cache!

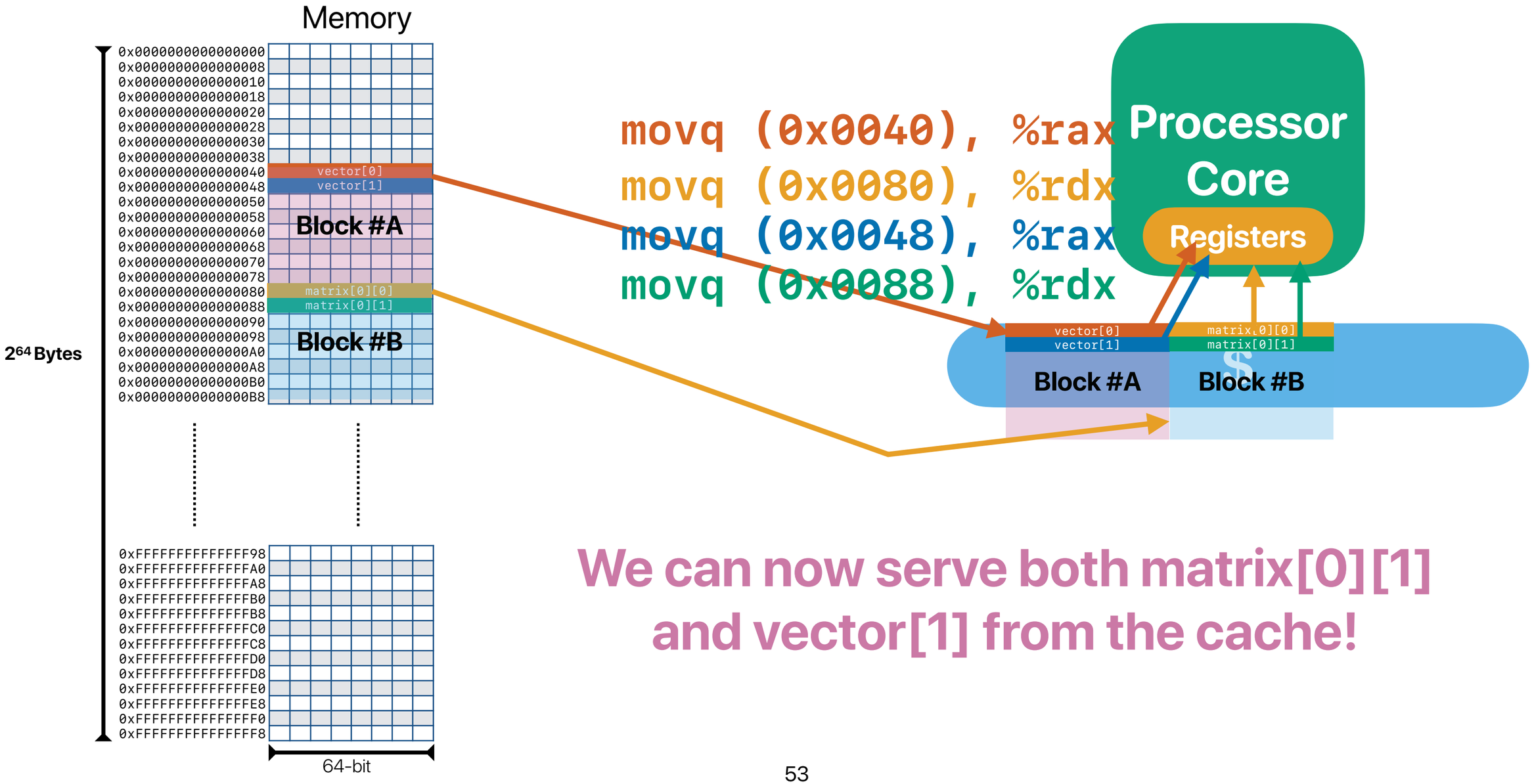
Recap: Locality

- Which description about locality of arrays `matrix` and `vector` in the following code is the **most accurate**?

```
for(uint64_t i = 0; i < m; i++) {
     result = 0;
    for(uint64_t j = 0; j < n; j++) {
          result += matrix[i][j]*vector[j];
    }
     output[i] = result;
 }
```

Simply caching one block isn't enough

We need to cache multiple blocks of data!



We can now serve both matrix[0][1] and vector[1] from the cache!

Designing a hardware to exploit locality

- Spatial locality — application tends to visit nearby stuffs in the memory

  We need to "cache consecutive memory locations" every time

  — the cache should store a "block" of code/data

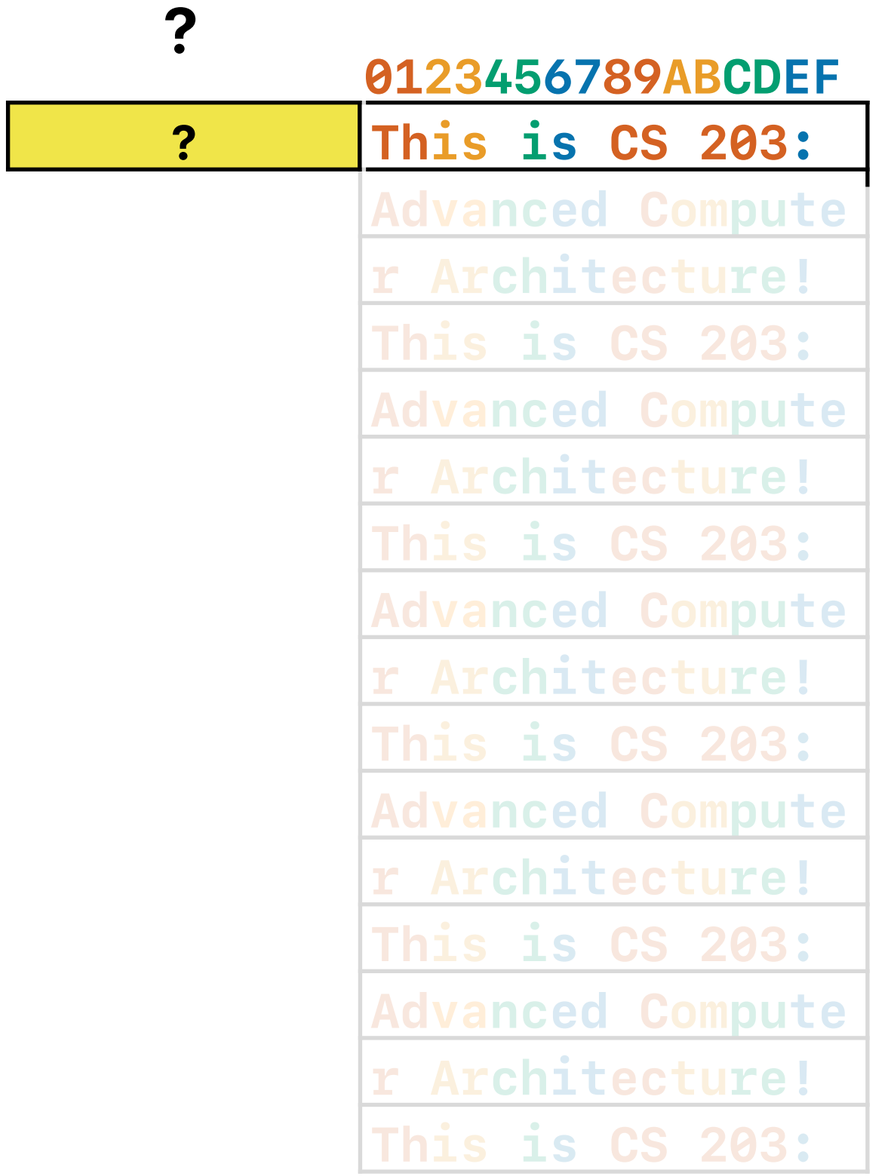
- Temporal locality — application revisit the same thing again and again

  We need to "cache frequently used memory blocks"

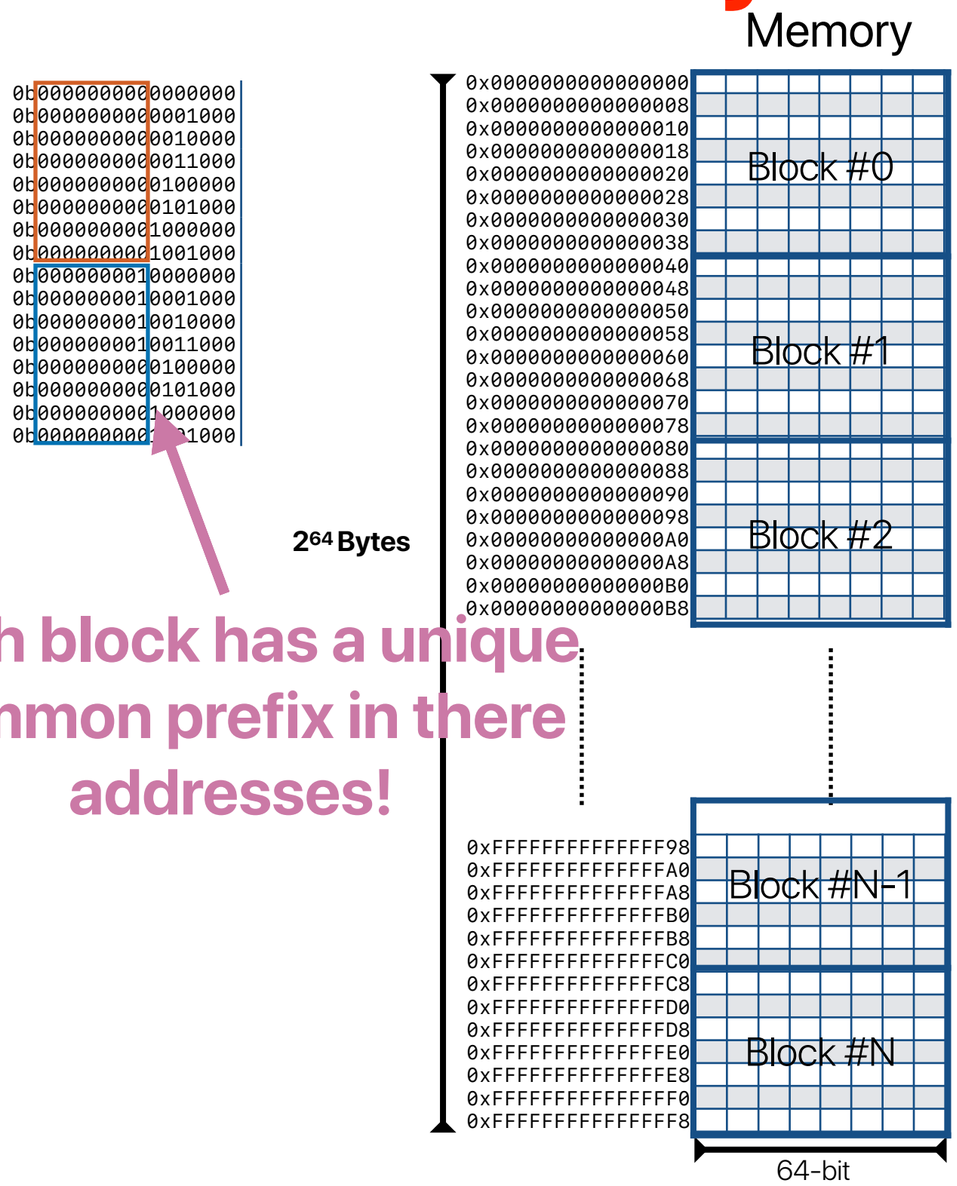
  — the cache should store a few blocks

  — the cache must be able to distinguish blocks

How to tell who is there?

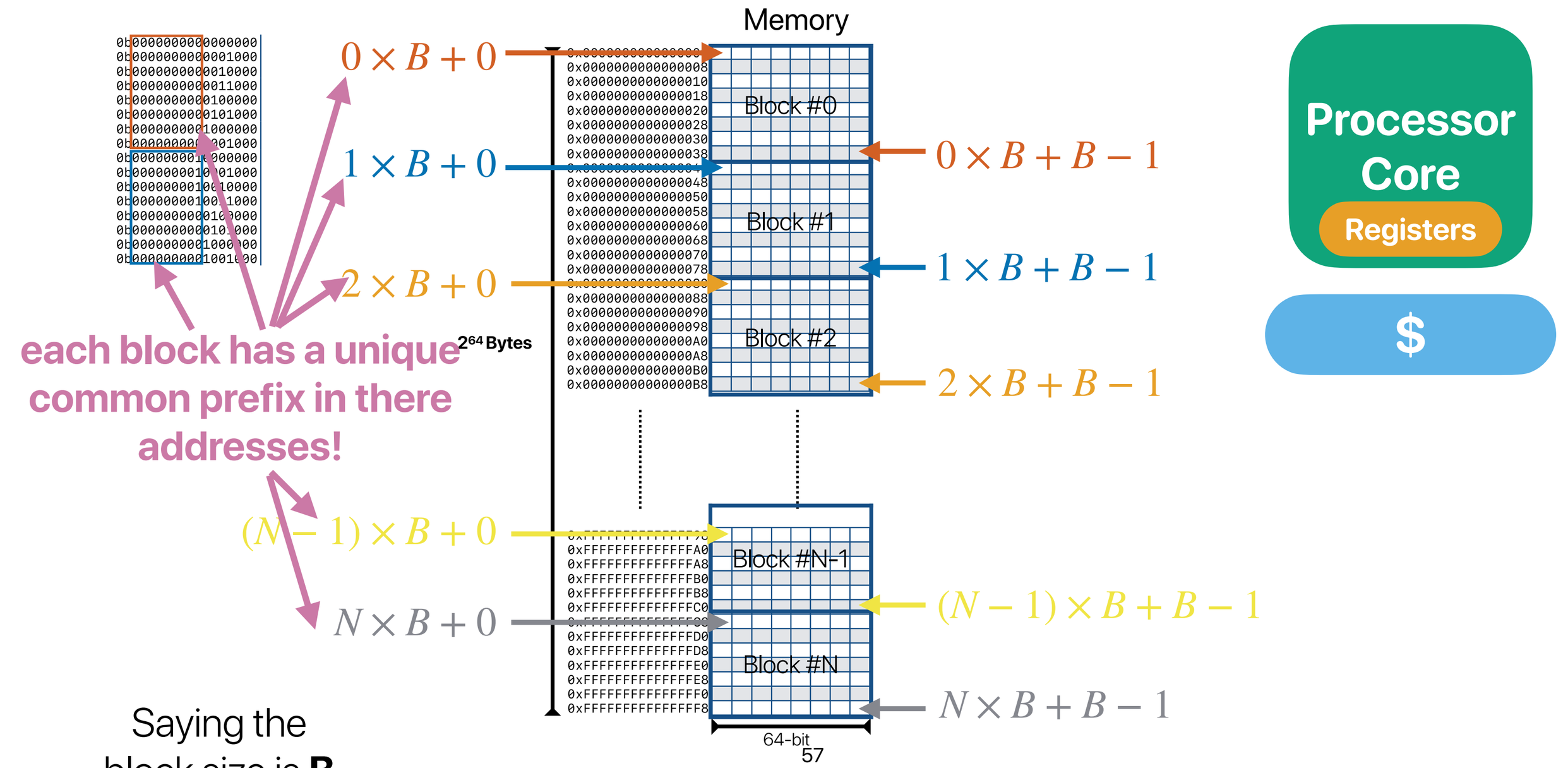


Partition memory addresses into fix-sized chunks



each block has a unique common prefix in there addresses!

Partition memory addresses into fix-sized chunks



Saying the block size is **B**

How to tell who is there?

the common address prefix in each block

tag array	0123456789ABCDEF
0x000	This is CS 203:
0x001	Advanced Compute
0xF07	r Architecture!
0x100	This is CS 203:
0x310	Advanced Compute
0x450	r Architecture!
0x006	This is CS 203:
0x537	Advanced Compute
0x266	r Architecture!
0x307	This is CS 203:
0x265	Advanced Compute
0x80A	r Architecture!
0x620	This is CS 203:
0x630	Advanced Compute
0x705	r Architecture!
0x216	This is CS 203:

How to tell w

Processor Core

Registers

block offset

tag

lw 0x0008

lw 0x4048

0x404 not found,
go to lower-level memory

Valid Bit

Dirty Bit

Tell if the block here can be used

Tell if the block here is modified

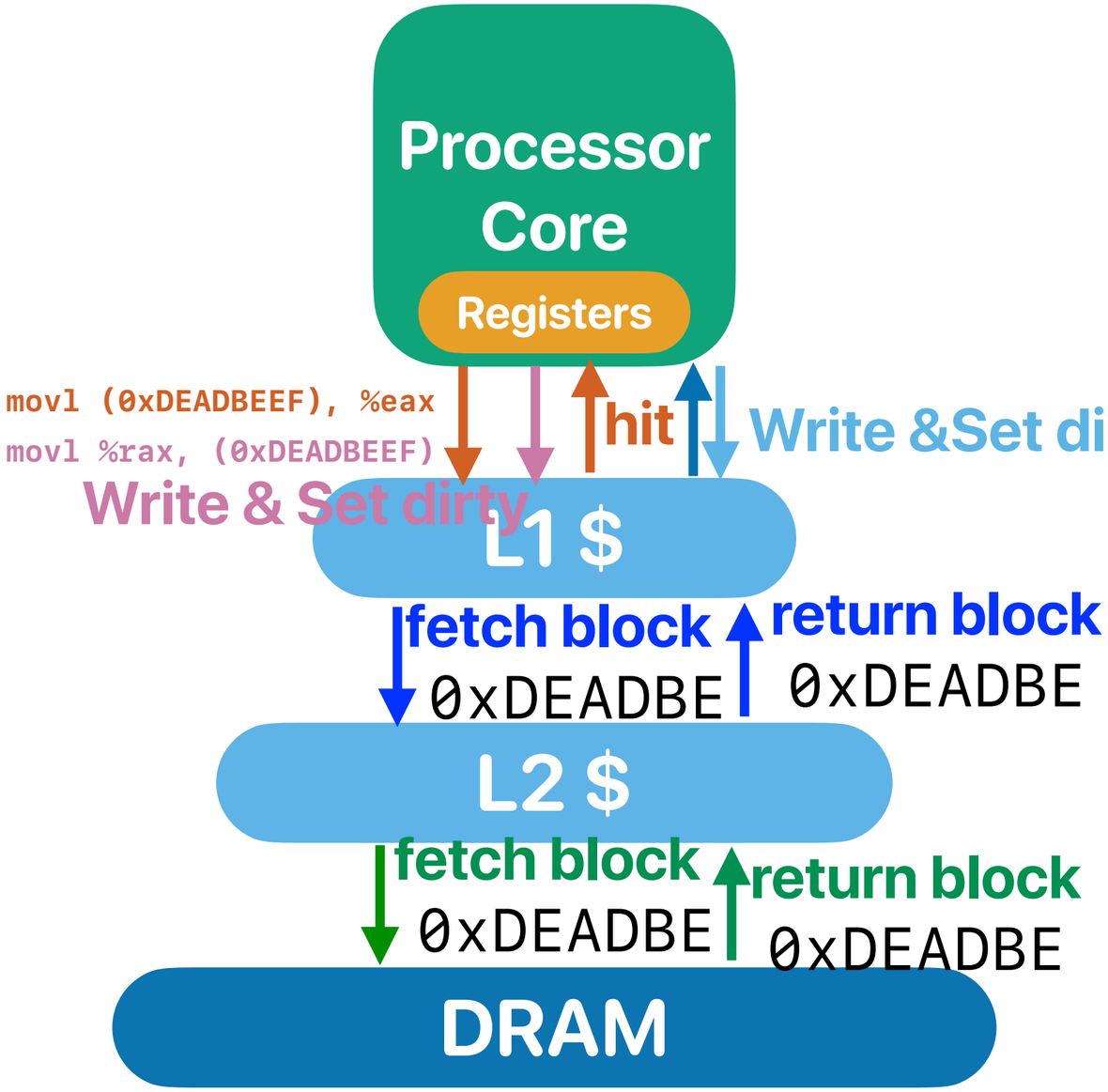
Valid Bit	Dirty Bit	tag	data (0123456789ABCDEF)
1	1	0x000	This is CSE1 3:
1	1	0x001	Advanced Compute
1	0	0xF07	r Architecture!
0	1	0x100	This is CS 203:
1	1	0x310	Advanced Compute
1	1	0x450	r Architecture!
0	1	0x006	This is CS 203:
0	1	0x537	Advanced Compute
1	1	0x266	r Architecture!
1	1	0x307	This is CS 203:
0	1	0x265	Advanced Compute
0	1	0x80A	r Architecture!
1	1	0x620	This is CS 203:
1	1	0x630	Advanced Compute
1	0	0x705	r Architecture!
0	1	0x216	This is CS 203:

Take-aways: designing caches

- Basic cache structures —
  - Caching in granularity of a block to capture spatial locality
  - Caching multiple blocks to keep frequently used data — temporal locality
  - Tags to distinguish cached blocks

Put everything all together: How cache interacts with CPU

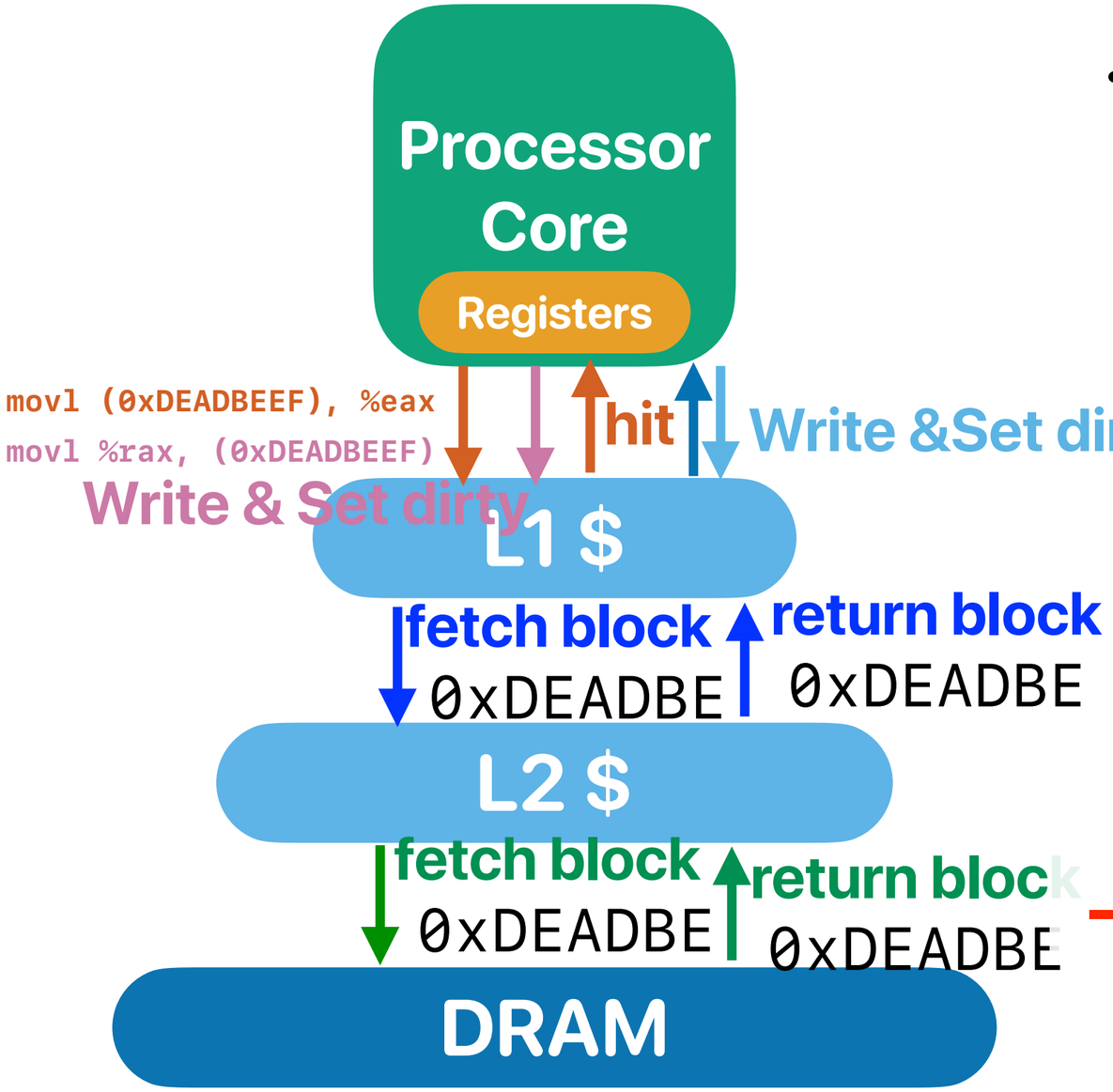
Processor/cache interaction



- Processor sends memory access request to L1-$
  - **if hit & it's a read**
    - **Read: return data**
    - **Write: Update "ONLY" in L1 and set DIRTY** **Why don't we write to L2? — Too slow**
  - **if miss**
    - Fetch the requesting block from lower-level memory hierarchy and place in the cache
    - **Present the write "ONLY" in L1 and set DIRTY**

What if we run out of $ blocks?

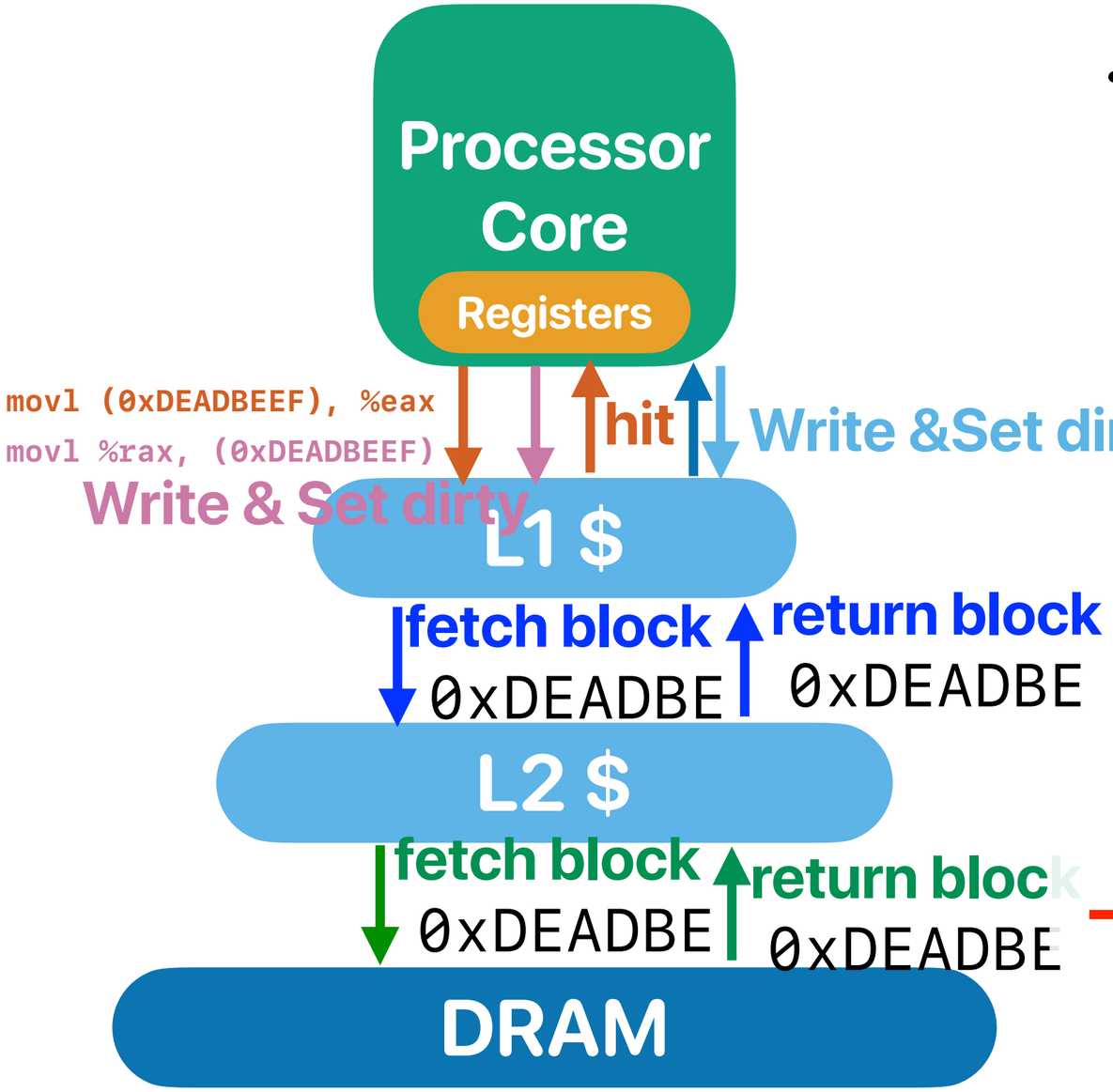
Processor/cache interaction



- Processor sends memory access request to L1-$
  - **if hit & it's a read**
    - **Read: return data**
    - **Write: Update "ONLY" in L1 and set DIRTY**
  - **if miss**
    - If there an empty block — place the data there
    - If NOT (most frequent case) — select a **victim block**
      - Least Recently Used (LRU) policy
    - Fetch the requesting block from lower level memory hierarchy and place in the cache
    - Present the write "ONLY" in L1 and set DIRTY

What if the victim block is modified? — ignoring the update is not acceptable!

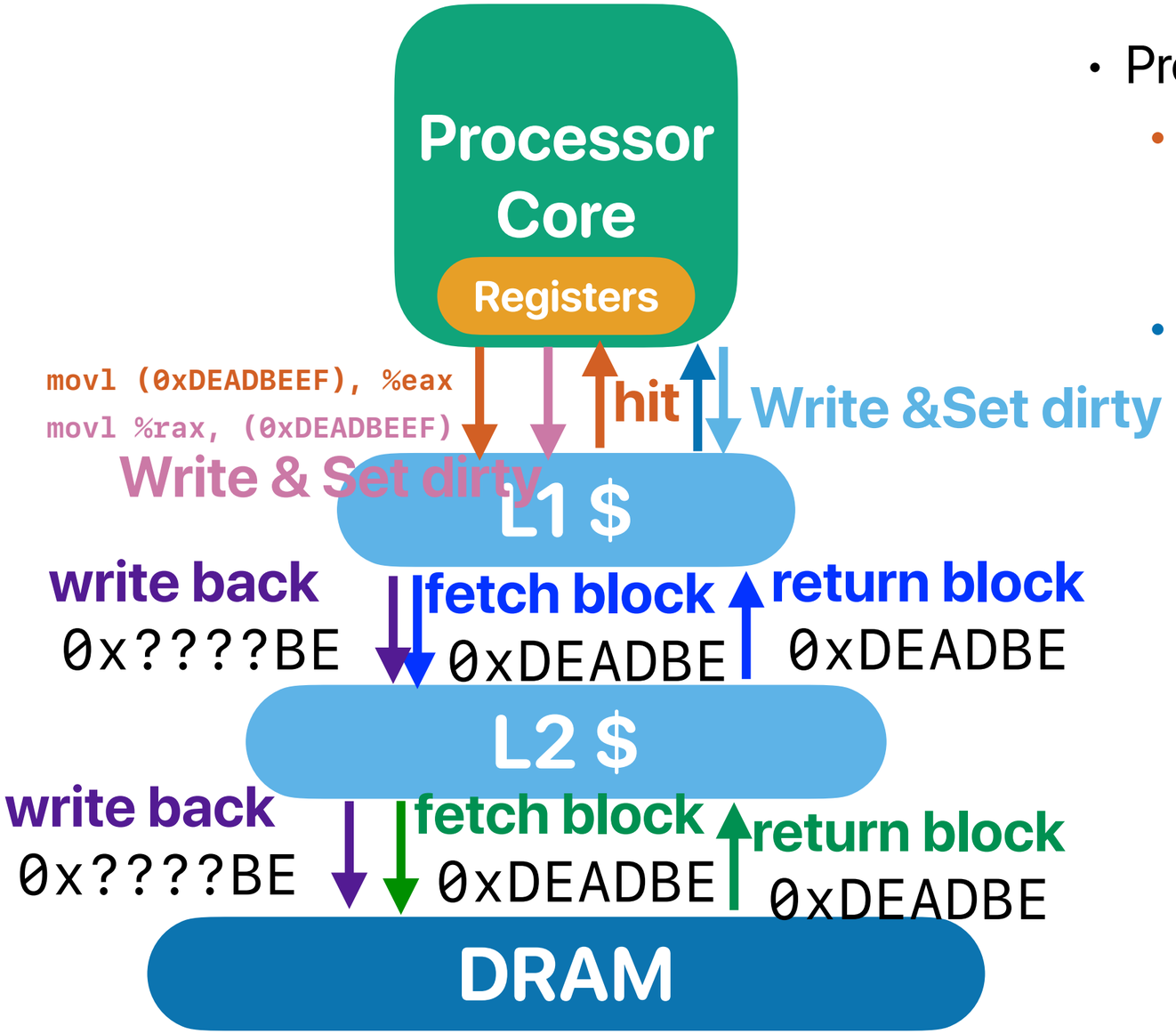
Processor/cache interaction



- Processor sends memory access request to L1-$
  - **if hit & it's a read**
    - **Read: return data**
    - **Write: Update "ONLY" in L1 and set DIRTY**
  - **if miss**
    - If there an empty block — place the data there
    - If NOT (most frequent case) — select a **victim block**
      - Least Recently Used (LRU) policy
    - Fetch the requesting block from lower level memory hierarchy and place in the cache
    - Present the write "ONLY" in L1 and set DIRTY

What if the victim block is modified?
— ignoring the update is not acceptable!

Processor/cache interaction

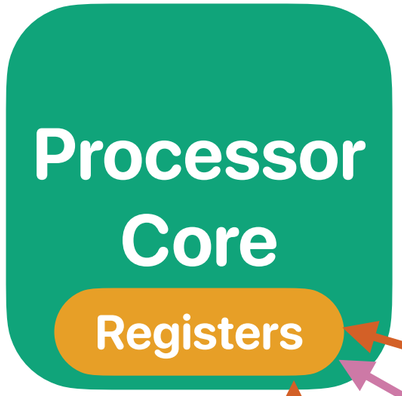


- Processor sends memory access request to L1-$
  - **if hit & it's a read**
    - **Read: return data**
    - **Write: Update "ONLY" in L1 and set DIRTY**
  - **if miss**
    - If there an empty block — place the data there
    - If NOT (most frequent case) — select a **victim block**
      - Least Recently Used (LRU) policy
    - If the victim block is "dirty" & "valid"
      - **Write back** the block to lower-level memory hierarchy
      - If write-back or fetching causes any miss, repeat the same process
    - Fetch the requesting block from lower-level memory hierarchy and place in the cache
    - **Present the write "ONLY" in L1 and set DIRTY**

Take-aways: designing caches

- Basic cache structures —
  - Caching in granularity of a block to capture spatial locality
  - Caching multiple blocks to keep frequently used data — temporal locality
  - Tags to distinguish cached blocks
- Hierarchical caching — data must be presented on the top level (L1) before the processor can use

How to tell w|



Tell if the block here can be used

Tell if the block here is modified

block offset

tag

lw 0x0008

lw 0x4048

0x404 not found,
go to lower-level memory

Valid Bit	Dirty Bit	tag	data 0123456789ABCDEF
1	1	0x000	This is CSE1 3:
1	1	0x001	Advanced Compute
1	0	0xF07	r Architecture!
0	1	0x100	This is CS 203:
1	1	0x310	Advanced Compute
1	1	0x450	r Architecture!
0	1	0x006	This is CS 203:
0	1	0x537	Advanced Compute
1	1	0x266	r Architecture!
1	1	0x307	This is CS 203:
0	1	0x265	Advanced Compute
0	1	0x80A	r Architecture!
1	1	0x620	This is CS 203:
1	1	0x630	Advanced Compute
1	0	0x705	r Architecture!
0	1	0x216	This is CS 203:

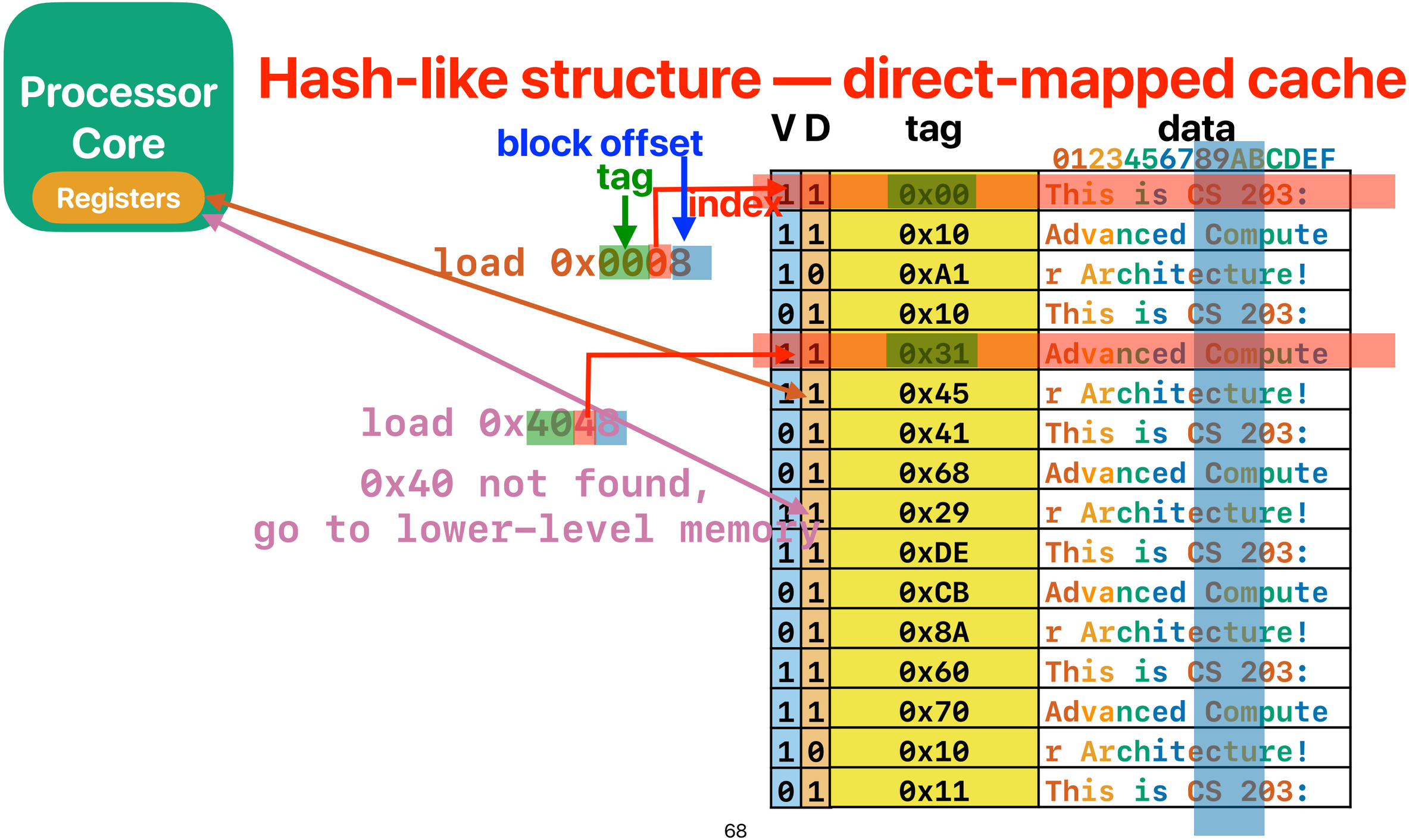
The complexity of search the matching tag—

$O(n)$— will be slow if our cache size grows!

Can we search things faster?

—hash table! $O(1)$

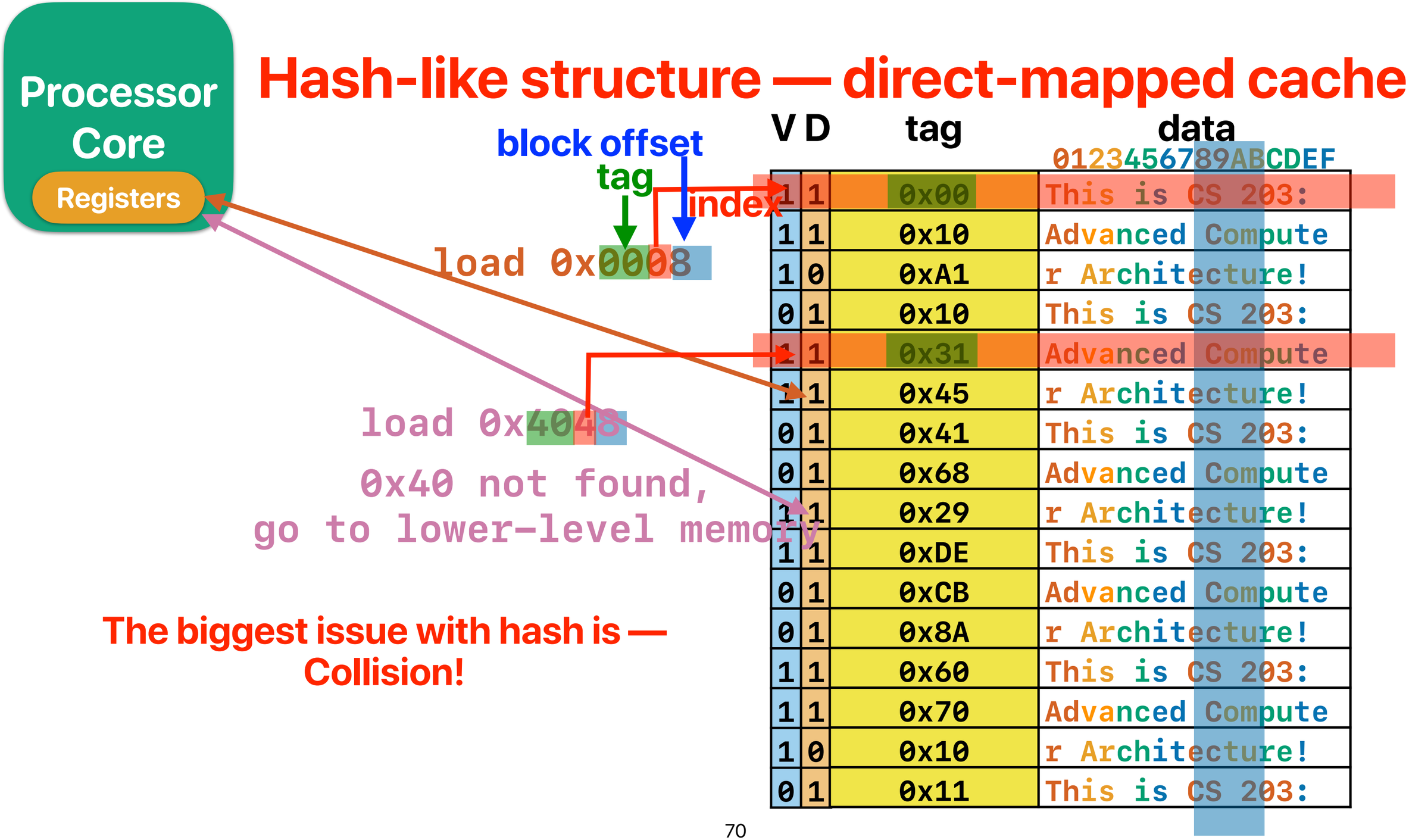
Hash-like structure — direct-mapped cache



Take-aways: designing caches

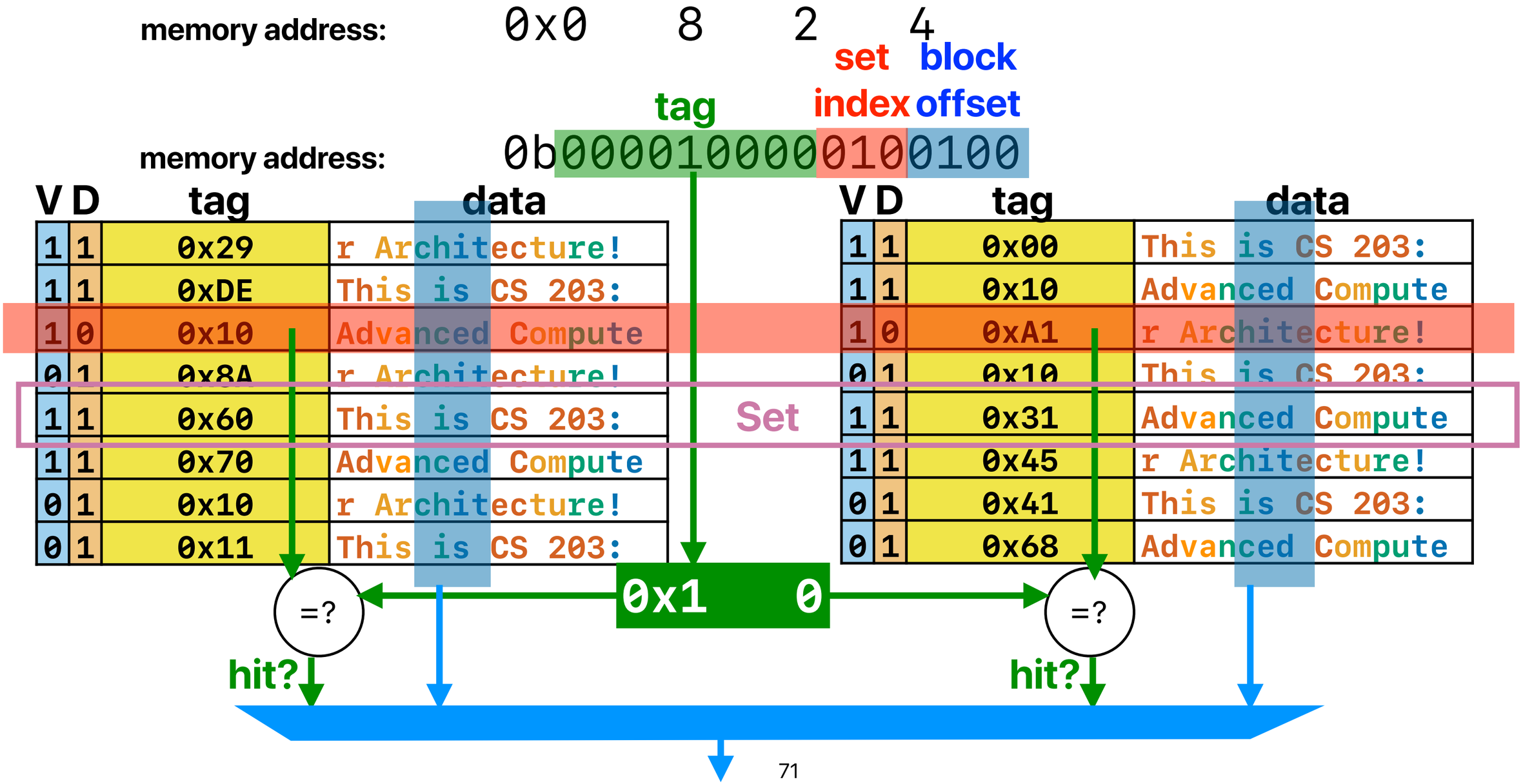
- Basic cache structures —
  - Caching in granularity of a block to capture spatial locality
  - Caching multiple blocks to keep frequently used data — temporal locality
  - Tags to distinguish cached blocks
- Hierarchical caching — data must be presented on the top level (L1) before the processor can use
- Optimizing cache structures
  - Hash block into “sets” to reduce the search time

Hash-like structure — direct-mapped cache



The biggest issue with hash is — Collision!

Way-associative cache



Blocksize == Linesize

```
/sys/devices/system/cpu/cpu0/cache/index0/physical_line_partition:1
/sys/devices/system/cpu/cpu0/cache/index0/number_of_sets:64
/sys/devices/system/cpu/cpu0/cache/index0/ways_of_associativity:12
/sys/devices/system/cpu/cpu0/cache/index0/id:0
/sys/devices/system/cpu/cpu0/cache/index0/shared_cpu_list:0-1
/sys/devices/system/cpu/cpu0/cache/index0/type:Data
/sys/devices/system/cpu/cpu0/cache/index0/size:48K
/sys/devices/system/cpu/cpu0/cache/index0/level:1
/sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size:64
/sys/devices/system/cpu/cpu0/cache/index0/shared_cpu_map:000003
```

Blocksize == Linesize

```
/sys/devices/system/cpu/cpu0/cache/index0/physical_line_partition:1
/sys/devices/system/cpu/cpu0/cache/index0/number_of_sets:64
/sys/devices/system/cpu/cpu0/cache/index0/ways_of_associativity:12
/sys/devices/system/cpu/cpu0/cache/index0/id:0
/sys/devices/system/cpu/cpu0/cache/index0/shared_cpu_list:0-1
/sys/devices/system/cpu/cpu0/cache/index0/type:Data
/sys/devices/system/cpu/cpu0/cache/index0/size:48K
/sys/devices/system/cpu/cpu0/cache/index0/level:1
/sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size:64
/sys/devices/system/cpu/cpu0/cache/index0/shared_cpu_map:000003
```

Take-aways: designing caches

- Basic cache structures —
  - Caching in granularity of a block to capture spatial locality
  - Caching multiple blocks to keep frequently used data — temporal locality
  - Tags to distinguish cached blocks
- Hierarchical caching — data must be presented on the top level (L1) before the processor can use
- Optimizing cache structures
  - Hash block into “sets” to reduce the search time
  - Set-associativity to reduce the “collision” problem

Way-associative cache

memory address: 0x0 8 2 4

set block

tag index offset

memory address: 0b000010000 010 0100

V	D	tag	data
1	1	0x29	r Architecture!
1	1	0xDE	This is CS 203:
1	0	0x10	Advanced Compute
0	1	0x8A	r Architecture!
1	1	0x60	This is CS 203:
1	1	0x70	Advanced Compute
0	1	0x10	r Architecture!
0	1	0x11	This is CS 203:

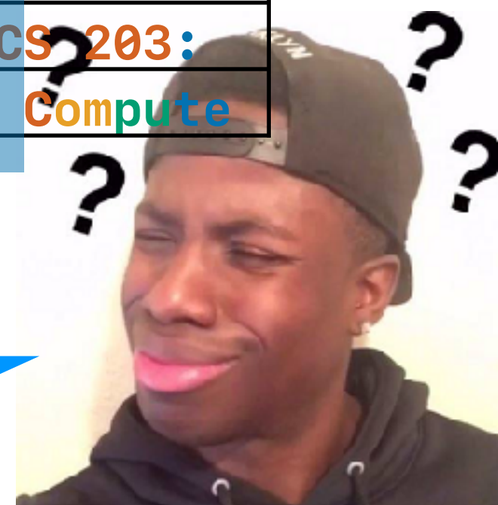
Set

V	D	tag	data
1	1	0x00	This is CS 203:
1	1	0x10	Advanced Compute
1	0	0xA1	r Architecture!
0	1	0x10	This is CS 203:
1	1	0x31	Advanced Compute
1	1	0x45	r Architecture!
0	1	0x41	This is CS 203:
0	1	0x68	Advanced Compute

0x1 0

=? =?

hit? hit?

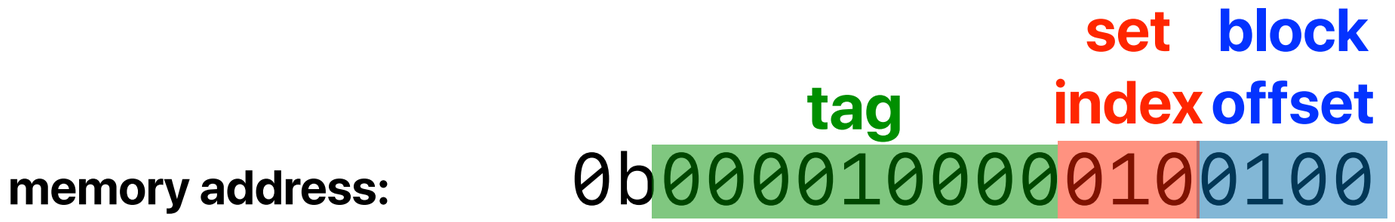


The A, B, Cs of your cache

C = ABS

- **C**: **C**apacity in data arrays
- **A**: Way-**A**ssociativity — how many blocks within a set
  - N-way: N blocks in a set, A = N
  - 1 for direct-mapped cache
- **B**: **B**lock Size (Linesize)
  - How many bytes in a block
- **S**: Number of **S**ets:
  - A set contains blocks sharing the same index
  - 1 for fully associate cache

Corollary of C = ABS



- number of bits in **b**lock offset — lg(**B**)
- number of bits in **s**et index: lg(**S**)
- tag bits: address_length - lg(S) - lg(B)
  - address_length is N bits for N-bit machines (e.g., 64-bit for 64-bit machines)
- (address / block_size) % S = set index

Take-aways: designing caches

- Basic cache structures —
  - Caching in granularity of a block to capture spatial locality
  - Caching multiple blocks to keep frequently used data — temporal locality
  - Tags to distinguish cached blocks
- Hierarchical caching — data must be presented on the top level (L1) before the processor can use
- Optimizing cache structures
  - Hash block into “sets” to reduce the search time
  - Set-associativity to reduce the “collision” problem
- C = A B S
  - C: capacity
  - A: Associativity
  - S: Number of sets
  - lg(S): Number of bits in set index
  - lg(B): Number of bits in block offset

Simulate the cache!

Simulate a direct-mapped cache

- A direct mapped (1-way) cache with 256 bytes total capacity, a block size of 16 bytes
  - # of blocks = $\frac{256}{16} = 16$
  - lg(16) = 4 : 4 bits are used for the index
  - lg(16) = 4 : 4 bits are used for the byte offset
  - The tag is 64 - (4 + 4) = 56 bits
  - For example: 0x 8 0 0 0 0 0 8 0
    = 0b1000 0000 0000 0000 0000 0000 1000 0000

Matrix vector revisited

```
for(uint64_t i = 0; i < m; i++) {
    result = 0;
    for(uint64_t j = 0; j < n; j++) {
         result += matrix[i][j]*vector[j];
    }
     output[i] = result;
}
```

Matrix vector revisited

```
for(uint64_t i = 0; i < m; i++) {
    result = 0;
    for(uint64_t j = 0; j < n; j++) {
          result += matrix[i][j]*vector[j];
    }
    output[i] = result;
}
```

tag index

tag index

	Address (Hex)	Address (Binary)
&a[0][0]	0x558FE0A1D330	0b101010110001111111000001010000111010011**00**110000
&b[0]	0x558FE0A1DC30	0b101010110001111111000001010000111011100**00**110000
&a[0][1]	0x558FE0A1D338	0b101010110001111111000001010000111010011**00**111000
&b[1]	0x558FE0A1DC38	0b101010110001111111000001010000111011100**00**111000
&a[0][2]	0x558FE0A1D340	0b101010110001111111000001010000111010011**01**000000
&b[2]	0x558FE0A1DC40	0b101010110001111111000001010000111011100**01**000000
&a[0][3]	0x558FE0A1D348	0b101010110001111111000001010000111010011**01**001000
&b[3]	0x558FE0A1DC48	0b101010110001111111000001010000111011100**01**001000
&a[0][4]	0x558FE0A1D350	0b101010110001111111000001010000111010011**01**010000
&b[4]	0x558FE0A1DC50	0b101010110001111111000001010000111011100**01**010000
&a[0][5]	0x558FE0A1D358	0b101010110001111111000001010000111010011**01**011000
&b[5]	0x558FE0A1DC58	0b101010110001111111000001010000111011100**01**011000
&a[0][6]	0x558FE0A1D360	0b101010110001111111000001010000111010011**01**100000
&b[6]	0x558FE0A1DC60	0b101010110001111111000001010000111011100**01**100000
&a[0][7]	0x558FE0A1D368	0b101010110001111111000001010000111010011**01**101000
&b[7]	0x558FE0A1DC68	0b101010110001111111000001010000111011100**01**101000
&a[0][8]	0x558FE0A1D370	0b101010110001111111000001010000111010011**01**110000
&b[8]	0x558FE0A1DC70	0b101010110001111111000001010000111011100**01**110000
&a[0][9]	0x558FE0A1D378	0b101010110001111111000001010000111010011**01**111000
&b[9]	0x558FE0A1DC78	0b101010110001111111000001010000111011100**01**111000

Simulate a direct-mapped cache

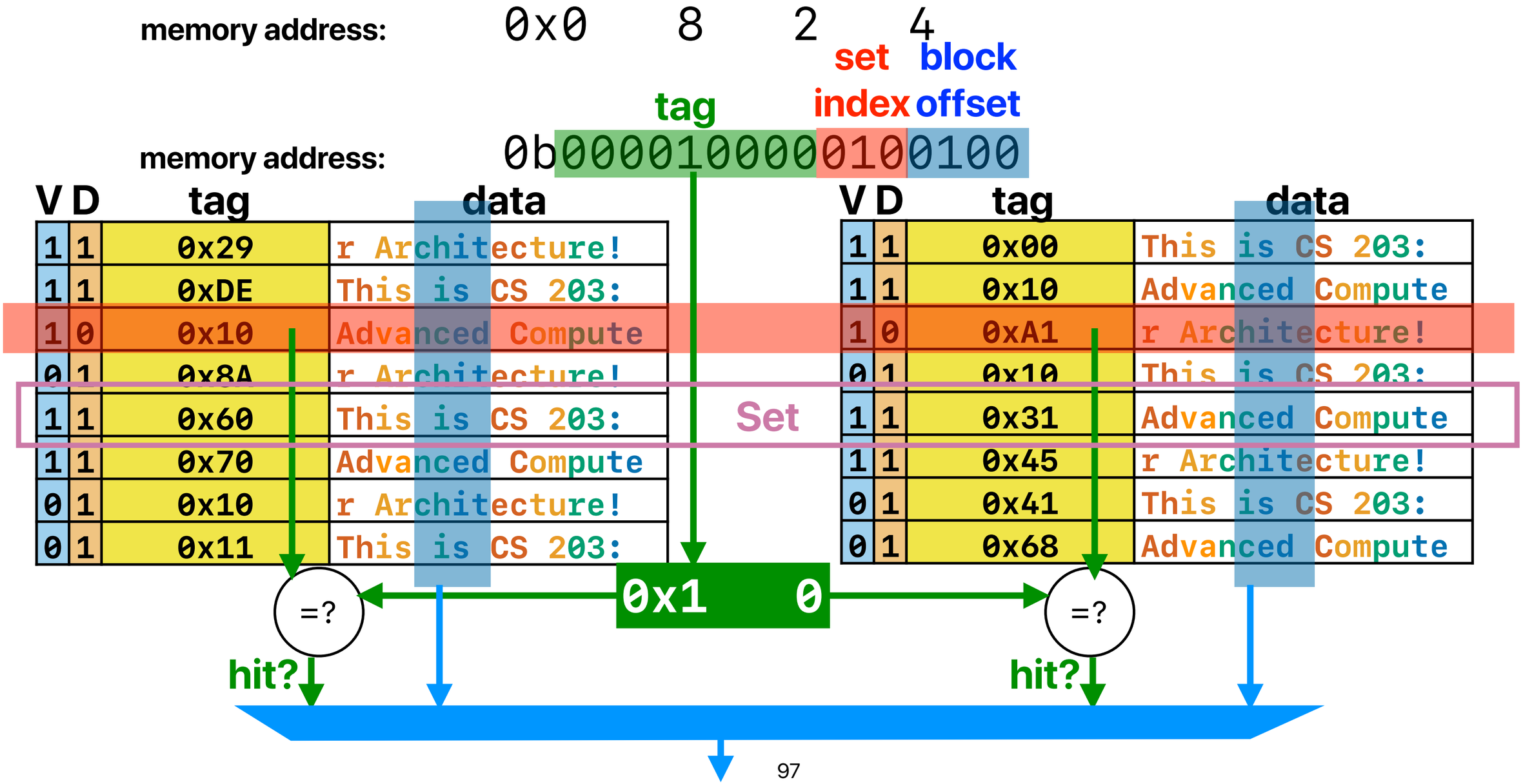
	V	D	Tag	Data
0	0	0		
1	0	0		
2	0	0		
3	1	0	0x558FE0A1DC	b[0], b[1]
4	1	0	0x558FE0A1DC	b[2], b[3]
5	0	0		
6	0	0		
7	0	0		
8	0	0		
9	0	0		
10	0	0		
11	0	0		
12	0	0		
13	0	0		
14	0	0		
15	0	0		

This cache doesn't work!!! — collisions!

tag index

	Address (Hex)	
&a[0][0]	0x558FE0A1D330	miss
&b[0]	0x558FE0A1DC30	miss
&a[0][1]	0x558FE0A1D338	miss
&b[1]	0x558FE0A1DC38	miss
&a[0][2]	0x558FE0A1D340	miss
&b[2]	0x558FE0A1DC40	miss
&a[0][3]	0x558FE0A1D348	miss
&b[3]	0x558FE0A1DC48	miss
&a[0][4]	0x558FE0A1D350	miss
&b[4]	0x558FE0A1DC50	miss
&a[0][5]	0x558FE0A1D358	miss
&b[5]	0x558FE0A1DC58	miss
&a[0][6]	0x558FE0A1D360	miss
&b[6]	0x558FE0A1DC60	miss
&a[0][7]	0x558FE0A1D368	miss
&b[7]	0x558FE0A1DC68	miss
&a[0][8]	0x558FE0A1D370	miss
&b[8]	0x558FE0A1DC70	miss
&a[0][9]	0x558FE0A1D378	
&b[9]	0x558FE0A1DC78	

Way-associative cache



Now, 2-way, same-sized cache

- A 2-way cache with 256 bytes total capacity, a block size of 16 bytes
  - # of blocks = $\frac{256}{16} = 16$
  - # of sets = $\frac{16}{2} = 8$ (2-way: 2 blocks in a set)
  - lg(8) = 3 : 3 bits are used for the index
  - lg(16) = 4 : 4 bits are used for the byte offset
  - The tag is 64 - (4 + 4) = 56 bits
  - For example: 0x 8 0 0 0 0 0 8 0
    = 0b1000 0000 0000 0000 0000 0000 1000 0000

Matrix vector revisited

```
for(uint64_t i = 0; i < m; i++) {
    result = 0;
    for(uint64_t j = 0; j < n; j++) {
        result += matrix[i][j]*vector[j];
    }
    output[i] = result;
}
```

tag index tag index

	Address (Hex)	Address (Binary)
&a[0][0]	0x558FE0A1D330	0b101010110001111111000001010000111010011**0**0110000
&b[0]	0x558FE0A1DC30	0b101010110001111111000001010000111011100**0**0110000
&a[0][1]	0x558FE0A1D338	0b101010110001111111000001010000111010011**0**0111000
&b[1]	0x558FE0A1DC38	0b101010110001111111000001010000111011100**0**0111000
&a[0][2]	0x558FE0A1D340	0b101010110001111111000001010000111010011**0**1000000
&b[2]	0x558FE0A1DC40	0b101010110001111111000001010000111011100**0**1000000
&a[0][3]	0x558FE0A1D348	0b101010110001111111000001010000111010011**0**1001000
&b[3]	0x558FE0A1DC48	0b101010110001111111000001010000111011100**0**1001000
&a[0][4]	0x558FE0A1D350	0b101010110001111111000001010000111010011**0**1010000
&b[4]	0x558FE0A1DC50	0b101010110001111111000001010000111011100**0**1010000
&a[0][5]	0x558FE0A1D358	0b101010110001111111000001010000111010011**0**1011000
&b[5]	0x558FE0A1DC58	0b101010110001111111000001010000111011100**0**1011000
&a[0][6]	0x558FE0A1D360	0b101010110001111111000001010000111010011**0**1100000
&b[6]	0x558FE0A1DC60	0b101010110001111111000001010000111011100**0**1100000
&a[0][7]	0x558FE0A1D368	0b101010110001111111000001010000111010011**0**1101000
&b[7]	0x558FE0A1DC68	0b101010110001111111000001010000111011100**0**1101000
&a[0][8]	0x558FE0A1D370	0b101010110001111111000001010000111010011**0**1110000
&b[8]	0x558FE0A1DC70	0b101010110001111111000001010000111011100**0**1110000
&a[0][9]	0x558FE0A1D378	0b101010110001111111000001010000111010011**0**1111000
&b[9]	0x558FE0A1DC78	0b101010110001111111000001010000111011100**0**1111000

Simulate a 2-way cache

V	D	Tag	Data	V	D	Tag	Data
0	0			0	0		
0	0			0	0		
0	0			0	0		
1	0	0xAB1FC143A6	a[0][0], a[0][1]	1	0	0xAB1FC143B8	b[0], b[1]
1	0	0xAB1FC143A6	a[0][2], a[0][3]	1	0	0xAB1FC143B8	b[2], b[3]
0	0			0	0		
0	0			0	0		
0	0			0	0		

	Address (Hex)	Tag	Index	
&a[0][0]	0x558FE0A1D330	0xAB1FC143A6	0x3	miss
&b[0]	0x558FE0A1DC30	0xAB1FC143B8	0x3	miss
&a[0][1]	0x558FE0A1D338	0xAB1FC143A6	0x3	hit
&b[1]	0x558FE0A1DC38	0xAB1FC143B8	0x3	hit
&a[0][2]	0x558FE0A1D340	0xAB1FC143A6	0x4	miss
&b[2]	0x558FE0A1DC40	0xAB1FC143B8	0x4	miss
&a[0][3]	0x558FE0A1D348	0xAB1FC143A6	0x4	hit
&b[3]	0x558FE0A1DC48	0xAB1FC143B8	0x4	hit
&a[0][4]	0x558FE0A1D350	0xAB1FC143A6	0x5	miss
&b[4]	0x558FE0A1DC50	0xAB1FC143B8	0x5	miss
&a[0][5]	0x558FE0A1D358	0xAB1FC143A6	0x5	hit
&b[5]	0x558FE0A1DC58	0xAB1FC143B8	0x5	hit
&a[0][6]	0x558FE0A1D360	0xAB1FC143A6	0x6	miss
&b[6]	0x558FE0A1DC60	0xAB1FC143B8	0x6	miss
&a[0][7]	0x558FE0A1D368	0xAB1FC143A6	0x6	hit
&b[7]	0x558FE0A1DC68	0xAB1FC143B8	0x6	hit
&a[0][8]	0x558FE0A1D370	0xAB1FC143A6	0x7	miss
&b[8]	0x558FE0A1DC70	0xAB1FC143B8	0x7	miss
&a[0][9]	0x558FE0A1D378	0xAB1FC143A6	0x7	hit
&b[9]	0x558FE0A1DC78	0xAB1FC143B8	0x7	hit

Estimating code performance on caches (cont.)

Taxonomy/reasons of cache misses

3Cs of misses

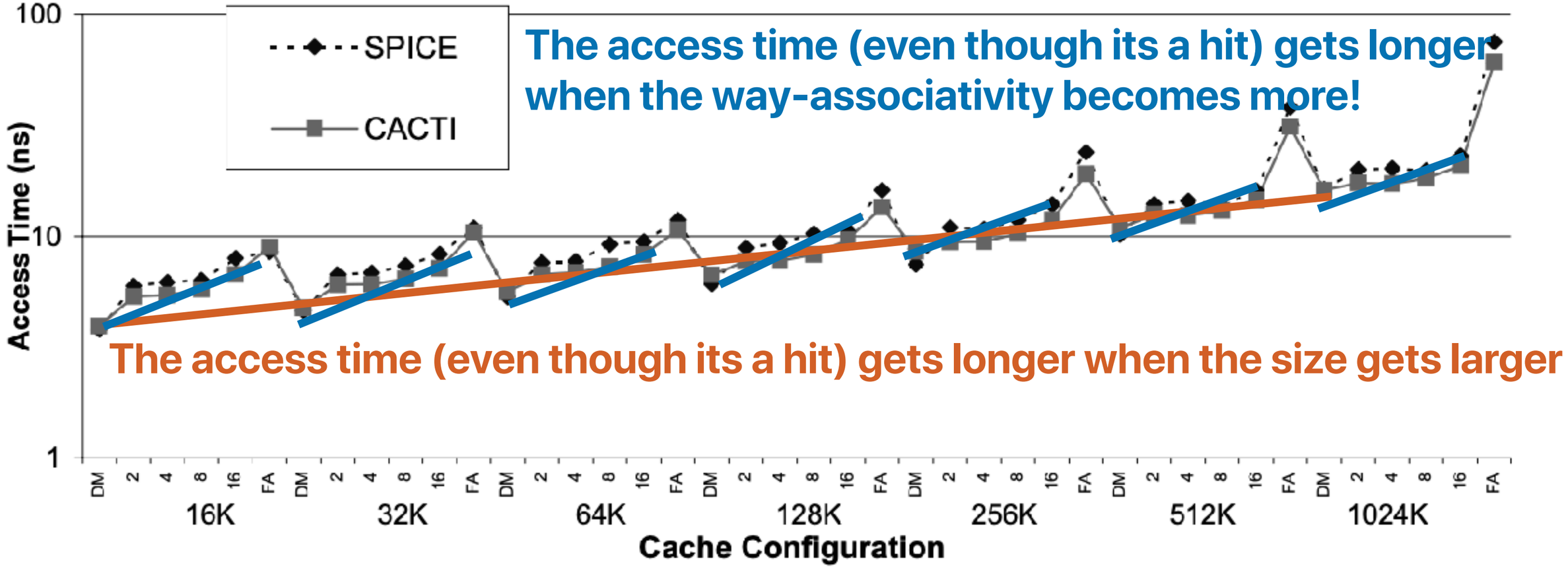
- Compulsory miss
  - Cold start miss. First-time access to a block
- Capacity miss
  - The working set size of an application is bigger than cache size
- Conflict miss
  - Required data replaced by block(s) mapping to the same set
  - Similar collision in hash

Take-aways: cache misses and their remedies

- Our code behaves differently on different cache configurations
- Cache misses
  - Compulsory misses — the miss due to first time access of a block
  - Conflict misses — the miss due to insufficient blocks of the target set **(associativity)**
  - Capacity misses — the miss due to the working set size surpasses the **capacity**

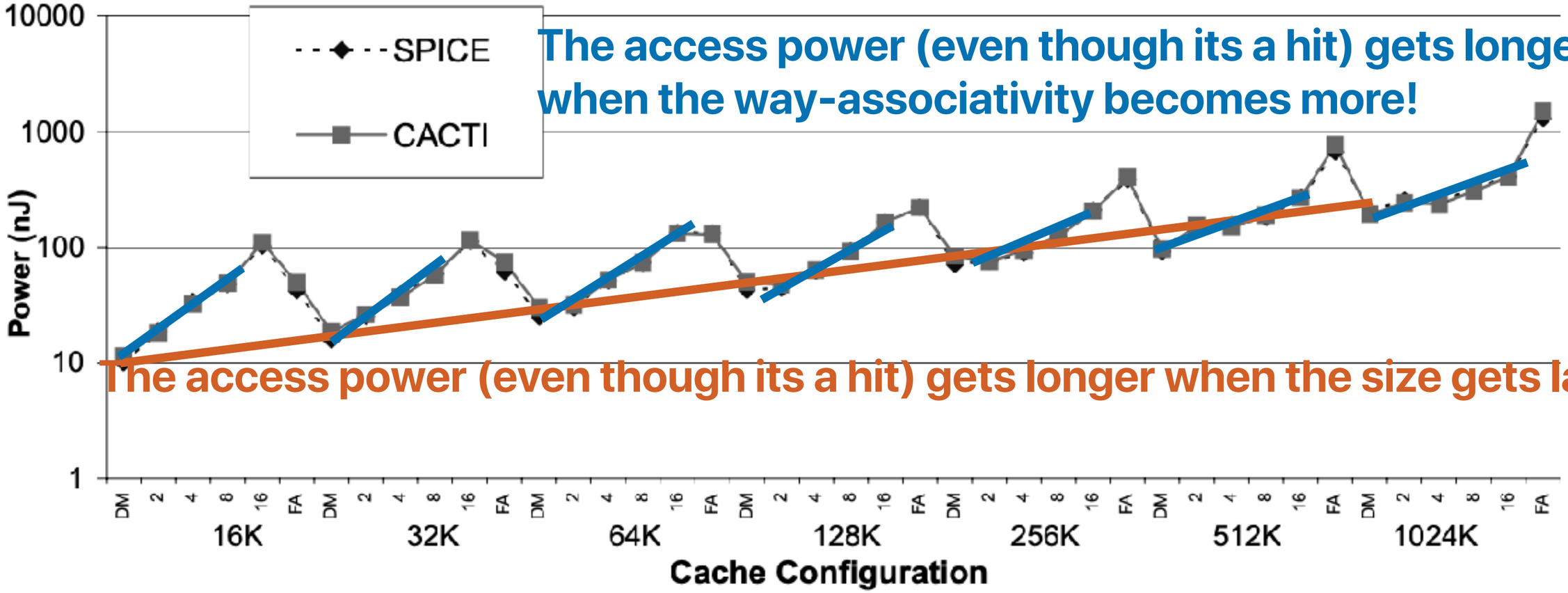
A, B, C, S and cache misses

Cache configurations and accessing time



Glen Reinman and Norman P. Jouppi. CACTI 2.0: An Integrated Cache Timing and Power Model. WRL Research Report 2000/7
https://bitsavers.trailing-edge.com/pdf/dec/tech_reports/WRL-2000-7.pdf

Cache configurations and accessing power



The access power (even though its a hit) gets longer when the way-associativity becomes more!

The access power (even though its a hit) gets longer when the size gets larger

Glen Reinman and Norman P. Jouppi. CACTI 2.0: An Integrated Cache Timing and Power Model. WRL Research Report 2000/7
https://bitsavers.trailing-edge.com/pdf/dec/tech_reports/WRL-2000-7.pdf

3Cs and A, B, C

- Regarding 3Cs: compulsory, conflict and capacity misses and A, B, C: associativity, block size, capacity
  How many of the following are correct?
  ① Without changing B & C, increasing A can reduce conflict misses but make each cache hit slower
  ② Without changing A & C, increasing B can reduce compulsory misses but potentially lead to more conflict misses
  ③ Without changing A & C, increasing B will make each cache miss slower
  ④ Without changing A & B, increasing C can reduce capacity misses but make each cache hit slower

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

You need to compare more tags

You reduce the number of sets now

You bring more into the cache when a miss occurs

Increases hit time because your data array is larger (longer time to fully charge your bit-lines)

You need to fetch more data for each miss

Take-aways: cache misses and their remedies

- Our code behaves differently on different cache configurations
- Cache misses
  - Compulsory misses — the miss due to first time access of a block
  - Conflict misses — the miss due to insufficient blocks of the target set **(associativity)**
  - Capacity misses — the miss due to the working set size surpasses the **capacity**
- There is no optimal cache configurations — trade-offs are everywhere
  - Increasing C — (+): capacity misses; (-): cost, access time, power
  - Increasing A — (+): conflict misses; (-): access time, power
  - Increasing B — (+): compulsory misses; (-): miss penalty

How can we improve cache performance without changing ABCs?

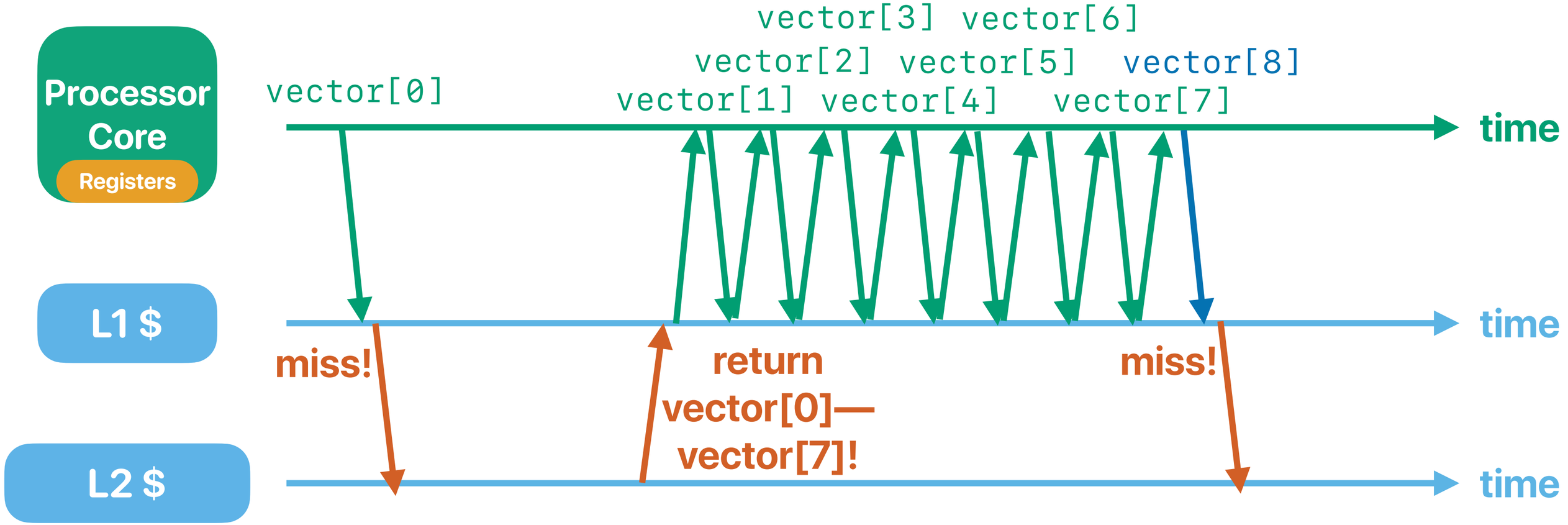
Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers

Norman P. Jouppi

Prefetching

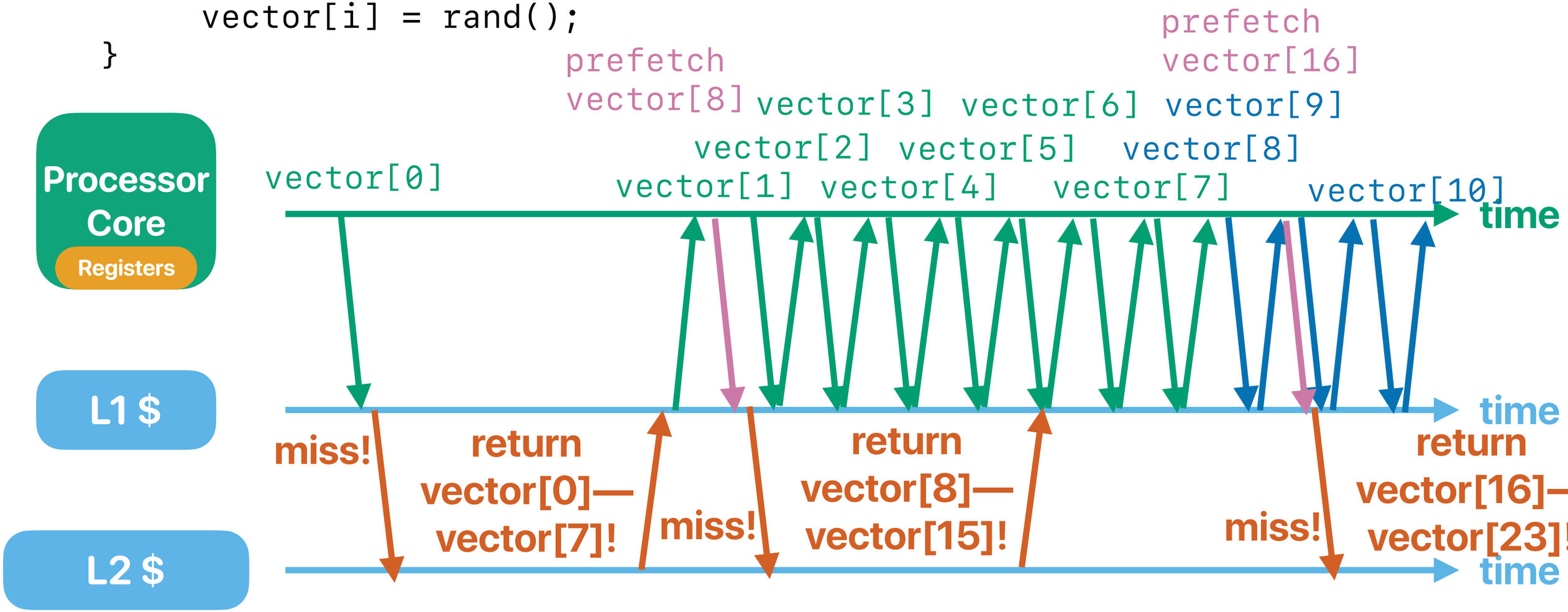
Spatial locality revisited

```
for(i = 0;i < size; i++) {
      vector[i] = rand();
}
```



What if we "pre-"fetch the next line?

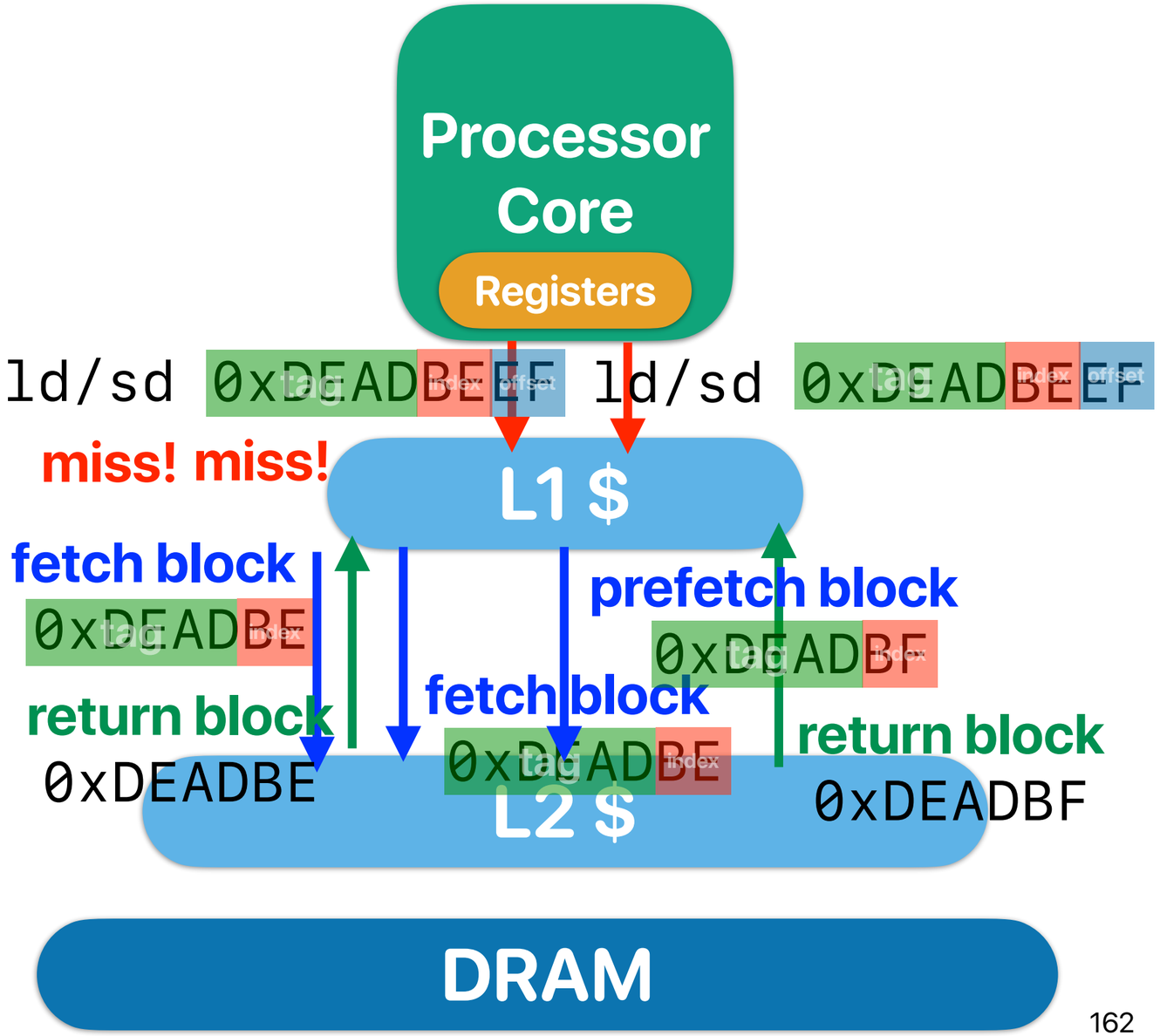
```
for(i = 0;i < size; i++) {
      vector[i] = rand();
}
```



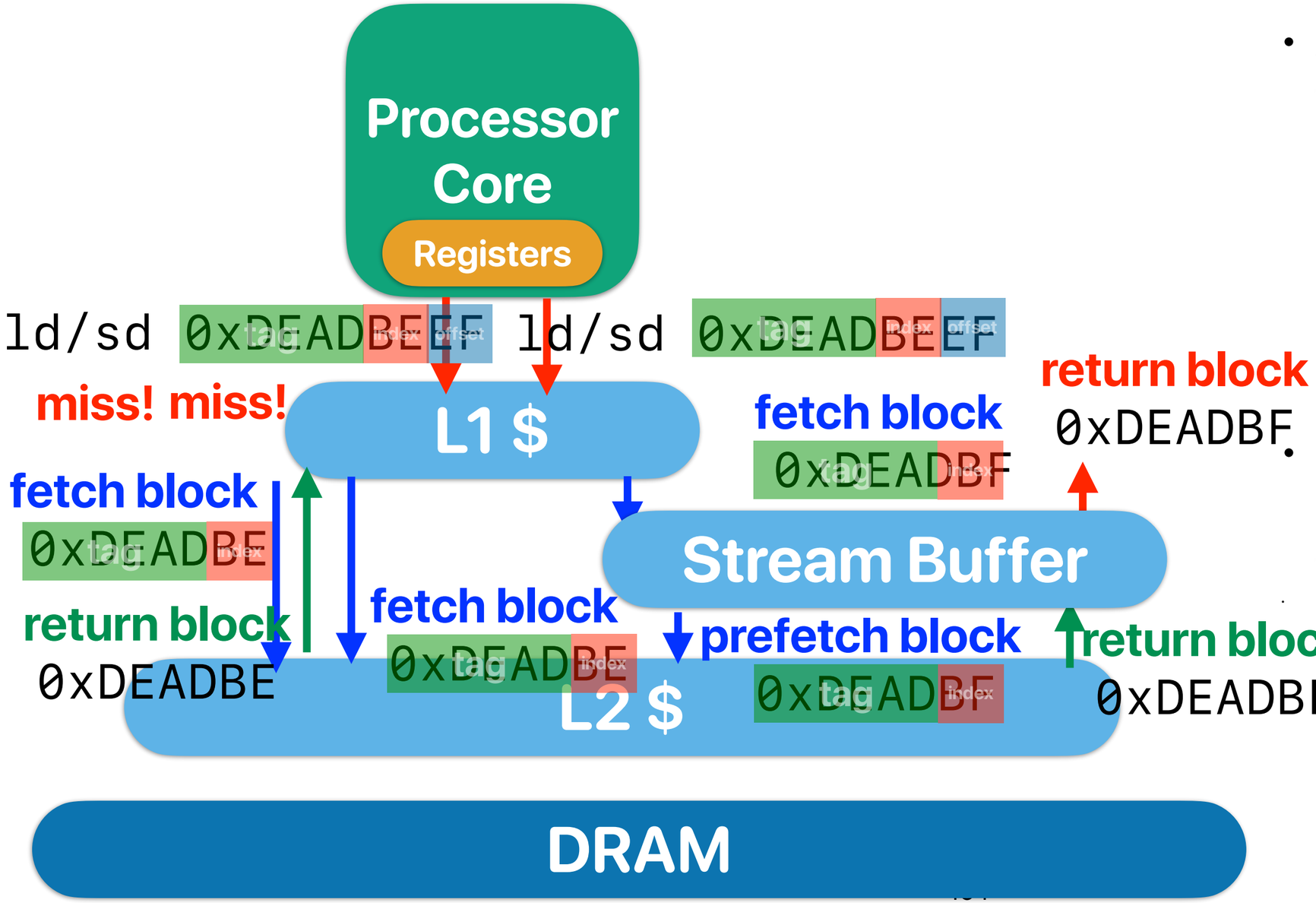
Hardware Prefetching

- The hardware identify the access pattern and proactively fetch data/instruction before the application asks for the data/instruction
- Trigger the cache miss earlier to eliminate the miss when the application needs the data/instruction
- The processor can keep track the distance between misses. If there is a pattern between misses, fetch `miss_data_address + offset` for a miss

What's after prefetching?



Stream buffer

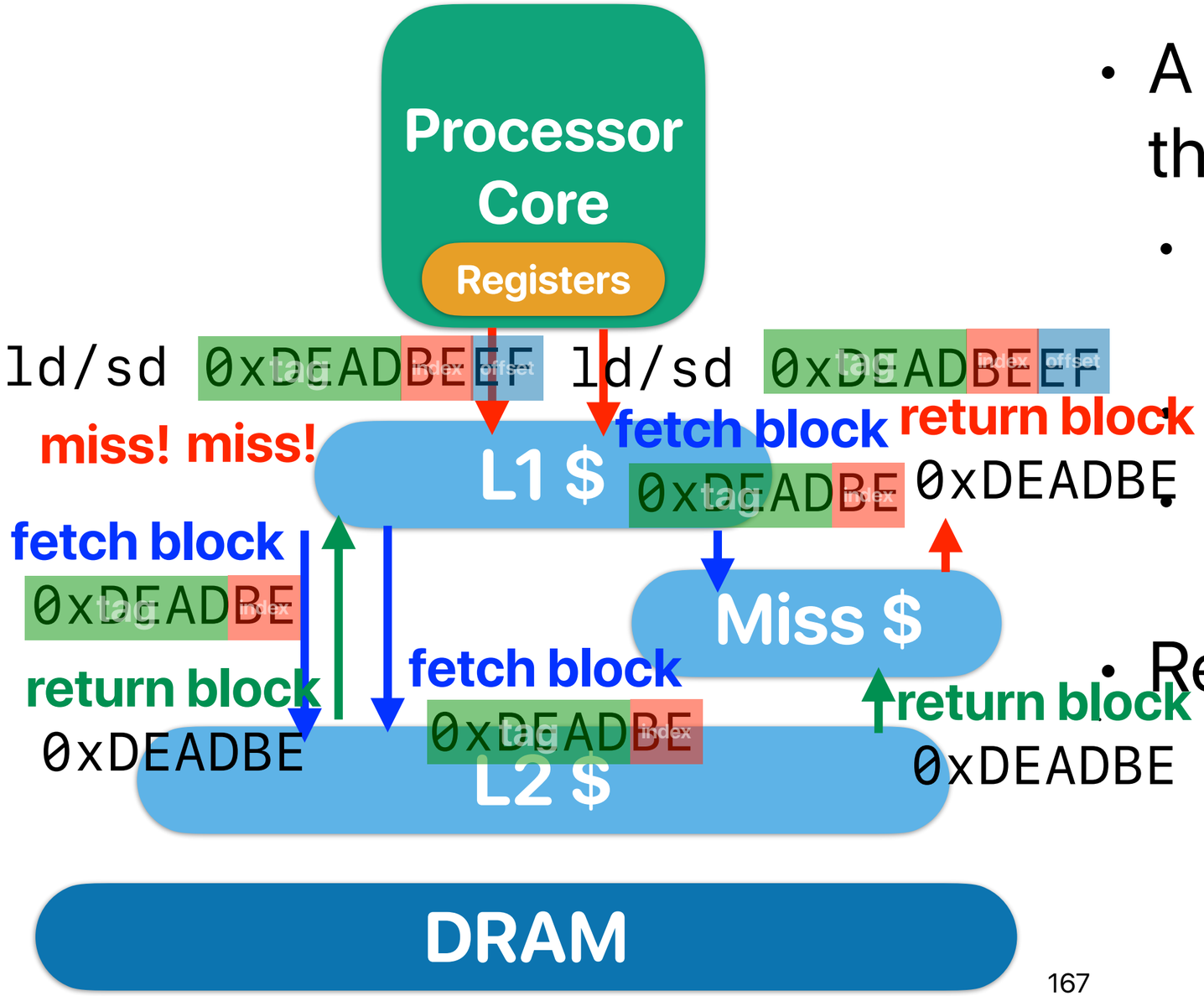


- A small cache that captures the prefetched blocks
  - Can be built as fully associative since it's small
  - Consult when there is a miss
  - Retrieve the block if found in the stream buffer
- Reduce compulsory misses and avoid conflict misses triggered by prefetching

Software Prefetching — through prefetching instructions

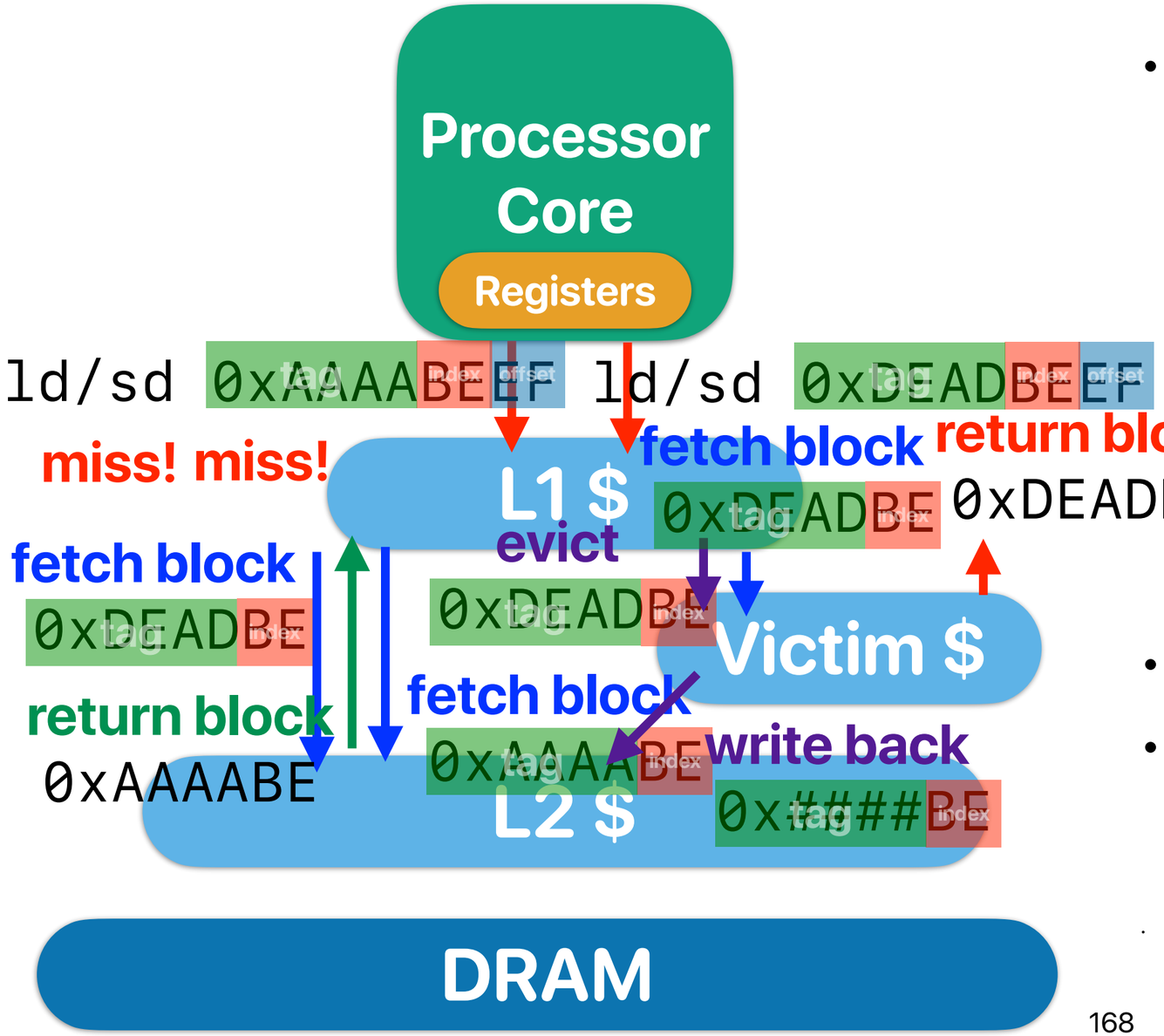
- x86 provide prefetch instructions
- As a programmer, you may insert `_mm_prefetch` in x86 programs to perform software prefetch for your code
- gcc also has a flag "-fprefetch-loop-arrays" to automatically insert software prefetch instructions

Miss cache



- A small cache that captures the missing blocks
  - Can be built as fully associative since it's small
  - Consult when there is a miss
  - Retrieve the block if found in the missing cache
- Reduce conflict misses

Victim cache



- A small cache that captures the evicted blocks
  - Can be built as fully associative since it's small
  - Consult when there is a miss
  - Swap the entry if hit in victim cache
  - Athlon/Phenom has an 8-entry victim cache
- Reduce conflict misses
- Jouppi [1990]: 4-entry victim cache removed 20% to 95% of conflicts for a 4 KB direct mapped data cache

Victim cache v.s. miss caching

- Both of them improves conflict misses
- Victim cache can use cache block more efficiently — swaps when miss
  - Miss caching maintains a copy of the missing data — the cache block can both in L1 and miss cache
  - Victim cache only maintains a cache block when the block is kicked out
- Victim cache captures conflict miss better
  - Miss caching captures every missing block

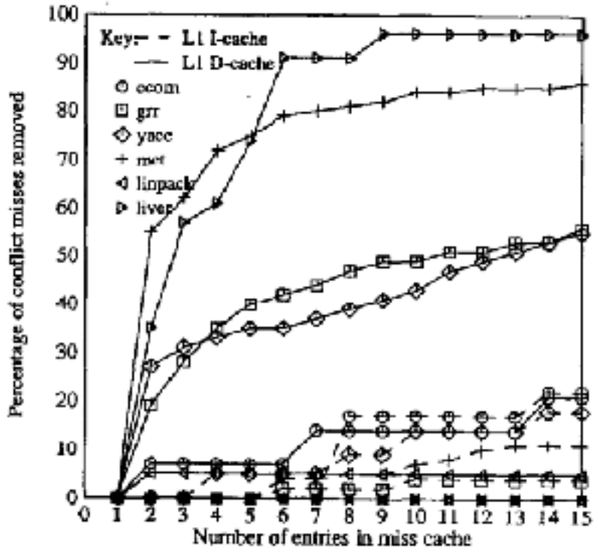


Figure 3-3: Conflict misses removed by miss caching

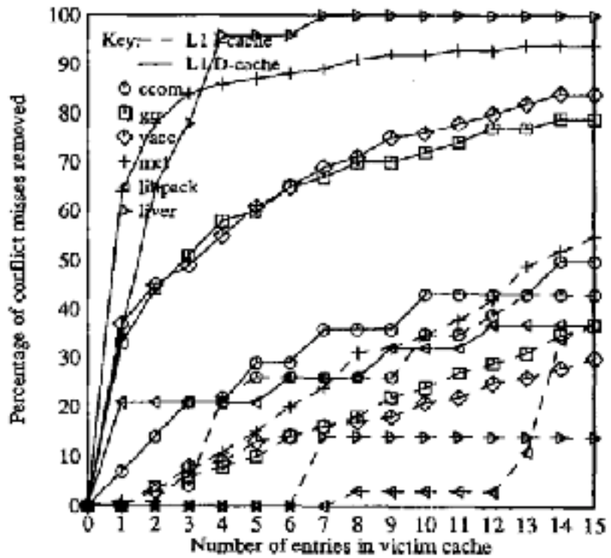


Figure 3-5: Conflict misses removed by victim caching

Takeaways: Optimizing cache performance through hardware

- There is no optimal cache configurations — trade-offs are everywhere
  - Increasing C — (+): capacity misses; (-): cost, access time, power
  - Increasing A — (+): conflict misses; (-): access time, power
  - Increasing B — (+): compulsory misses; (-): miss penalty
- Adding a small buffer alongside the L1 cache can —
  - Virtually add an associative set to frequently used data structures
  - Prefetched blocks won’t cause conflict misses

Takeaways: Optimizing cache performance through hardware

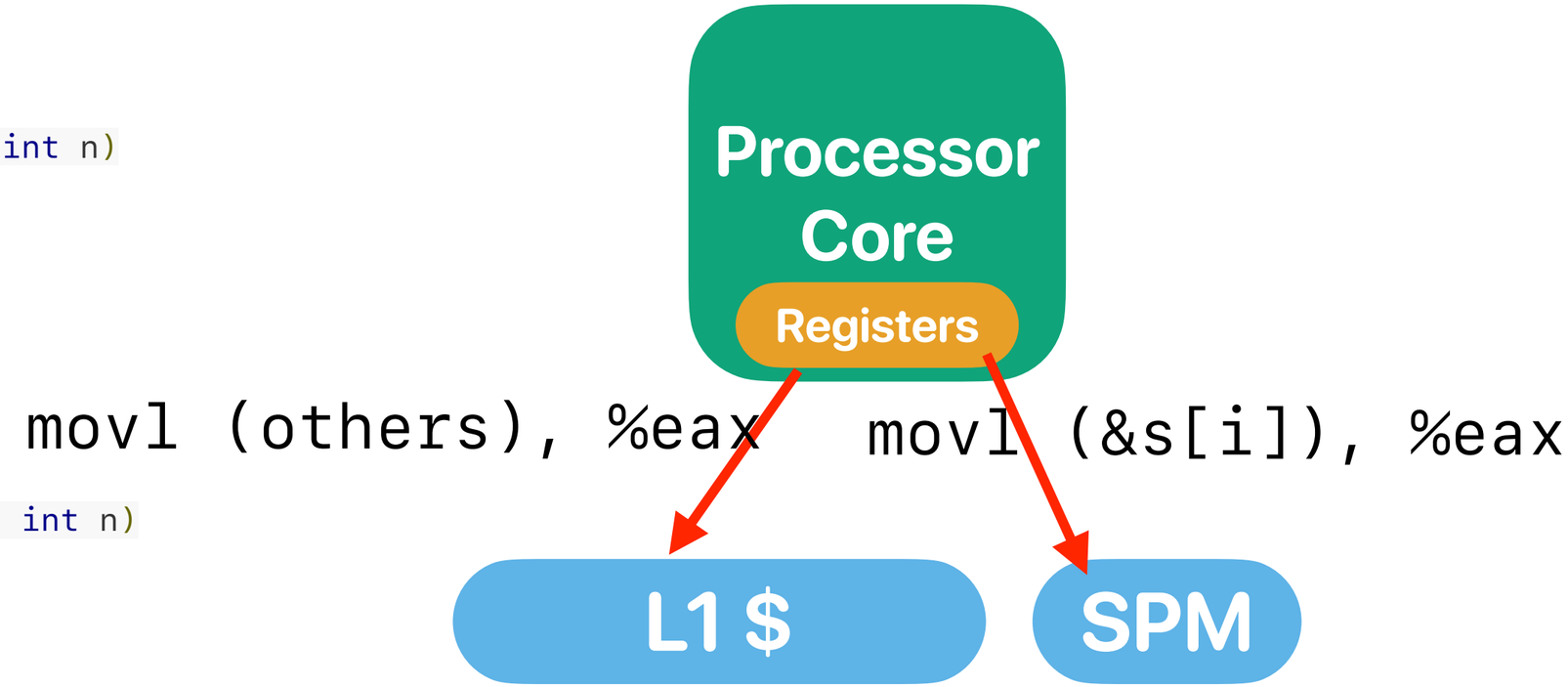
- There is no optimal cache configurations — trade-offs are everywhere
  - Increasing C — (+): capacity misses; (-): cost, access time, **power**
  - Increasing A — (+): conflict misses; (-): access time, **power**
  - Increasing B — (+): compulsory misses; (-): miss penalty
- Adding a small buffer alongside the L1 cache can —
  - Virtually add an associative set to frequently used data structures
  - Prefetched blocks won't cause conflict misses

We need additional search time and power

Implementation of SPM — GPU's shared memory

```
__global__ void staticReverse(int *d, int n)
{
  __shared__ int s[64];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}

__global__ void dynamicReverse(int *d, int n)
{
  extern __shared__ int s[];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}
```



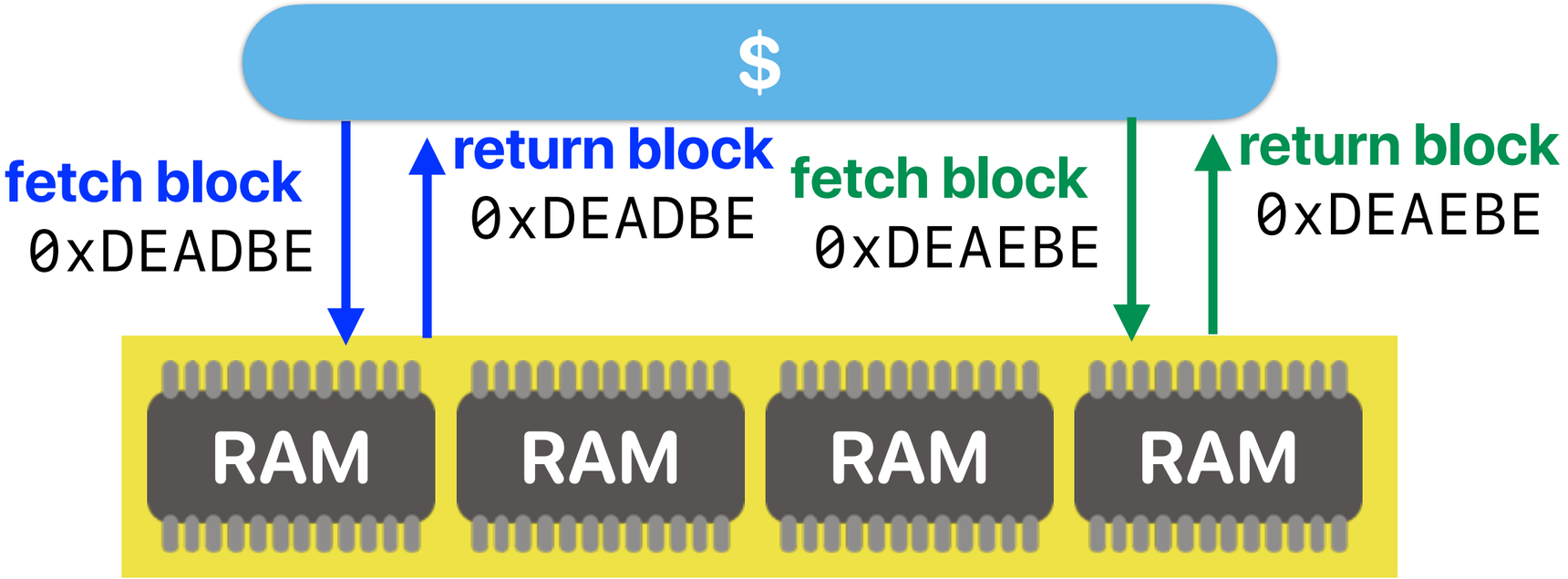
https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/

Takeaways: Optimizing cache performance through hardware

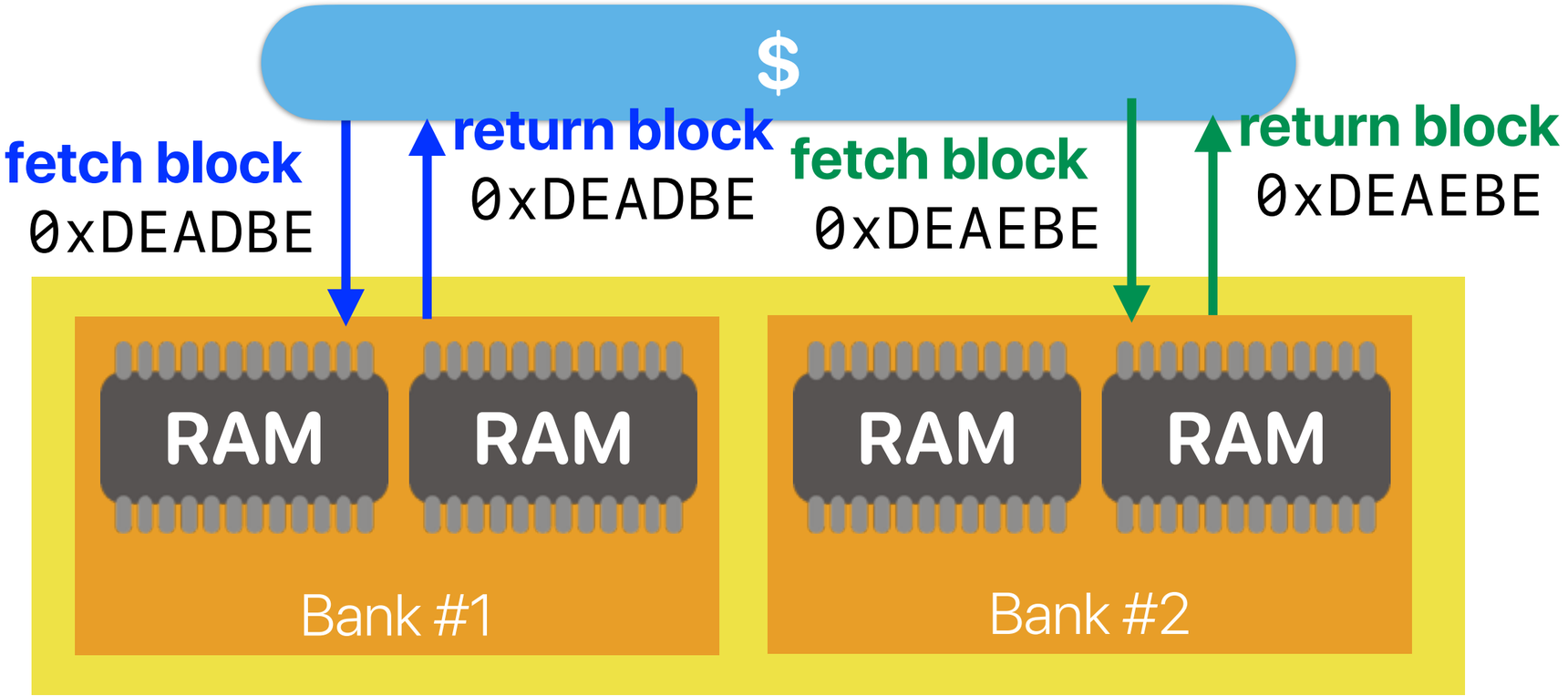
- There is no optimal cache configurations — trade-offs are everywhere
  - Increasing C — (+): capacity misses; (-): cost, access time, power
  - Increasing A — (+): conflict misses; (-): access time, power
  - Increasing B — (+): compulsory misses; (-): miss penalty
- Adding a small buffer alongside the L1 cache can —
  - Virtually add an associative set to frequently used data structures
  - Prefetched blocks won’t cause conflict misses
- Adding a tag-less, programmable small buffer alongside the L1 cache can reduce power consumption

Advanced Hardware Techniques in Improving Memory Performance

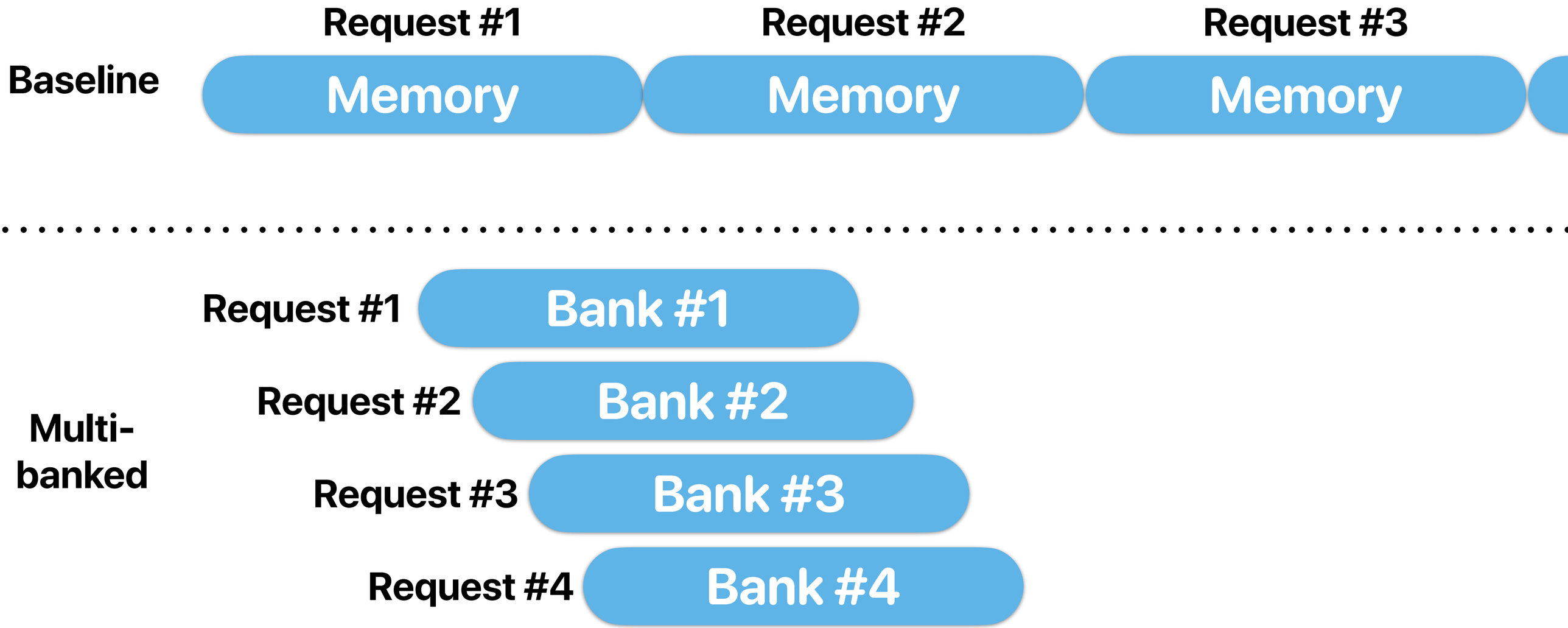
Blocking cache



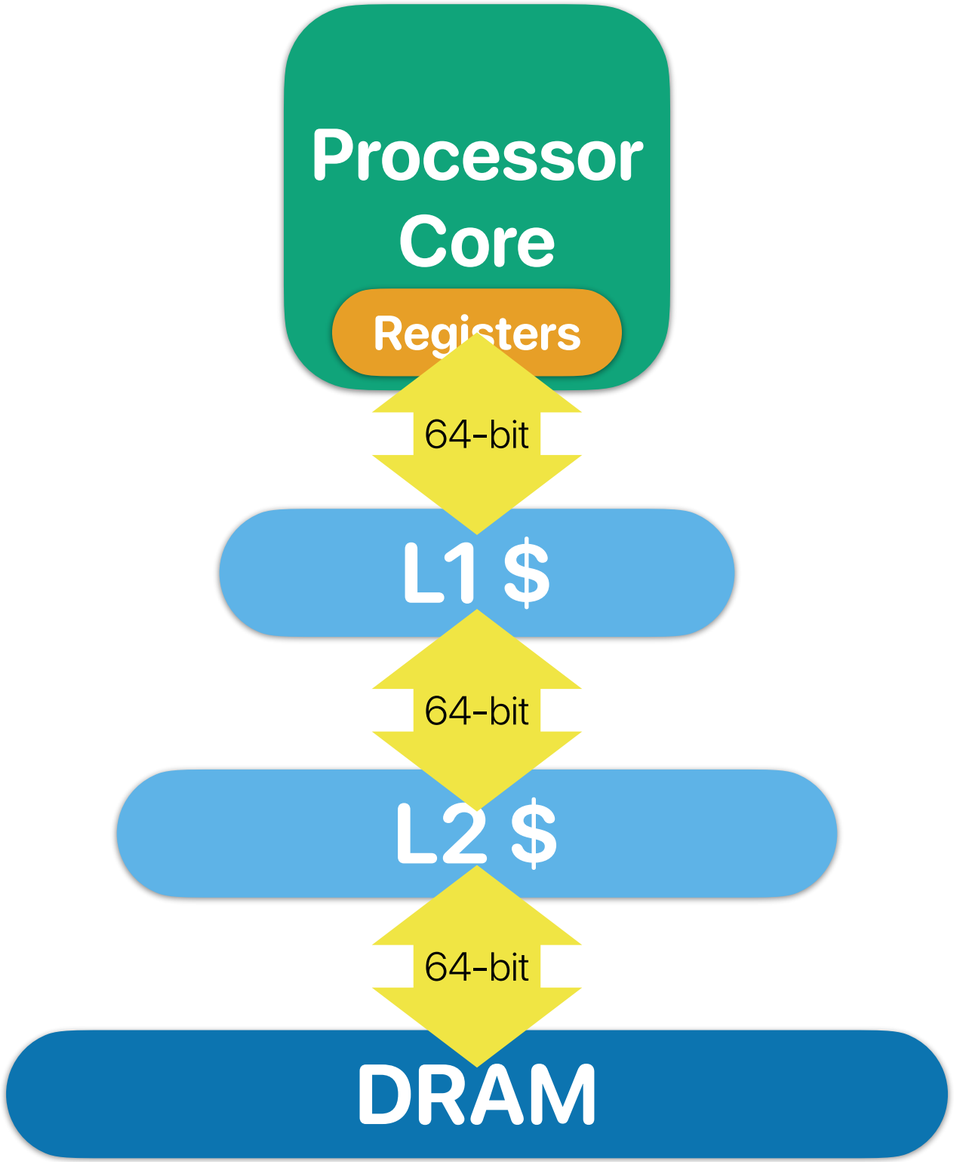
Multibanks & non-blocking caches



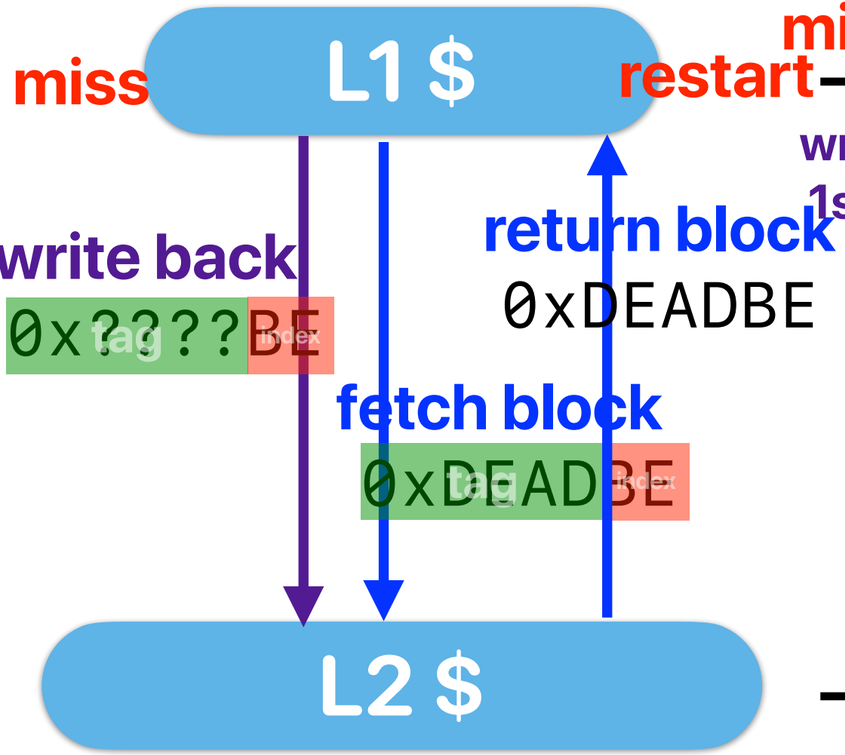
Pipelined access and multi-banked caches

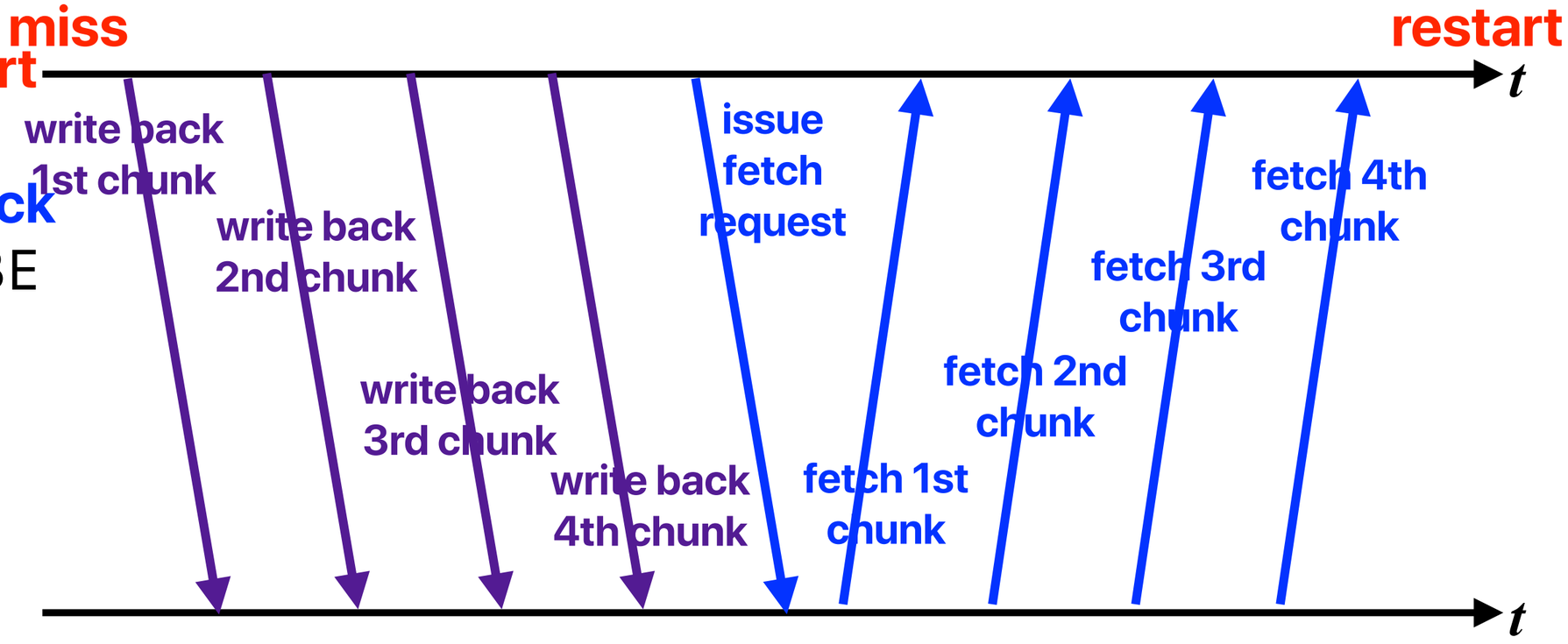


The bandwidth between units is limited



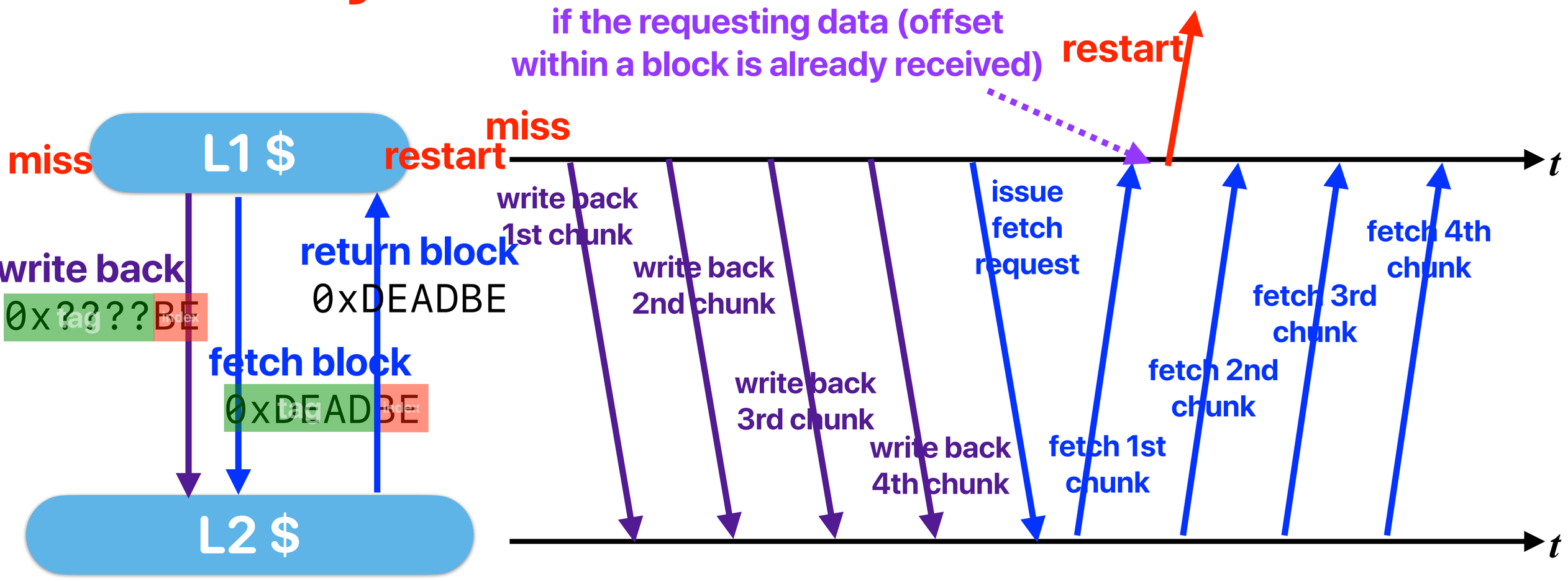
When we handle a miss





assume the bus between L1/L2 only allows a quarter of the cache block go through it

Early Restart and Critical Word First

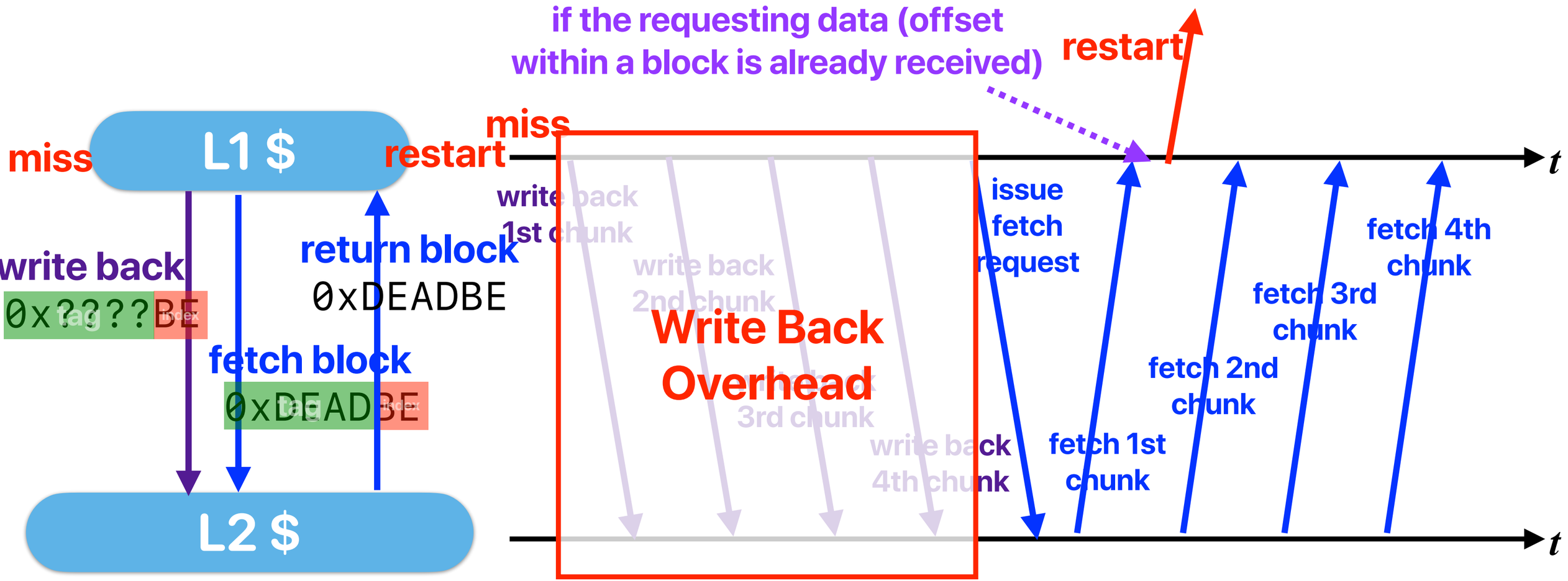


assume the bus between L1/L2 only allows a quarter of the cache block go through it

Early Restart and Critical Word First

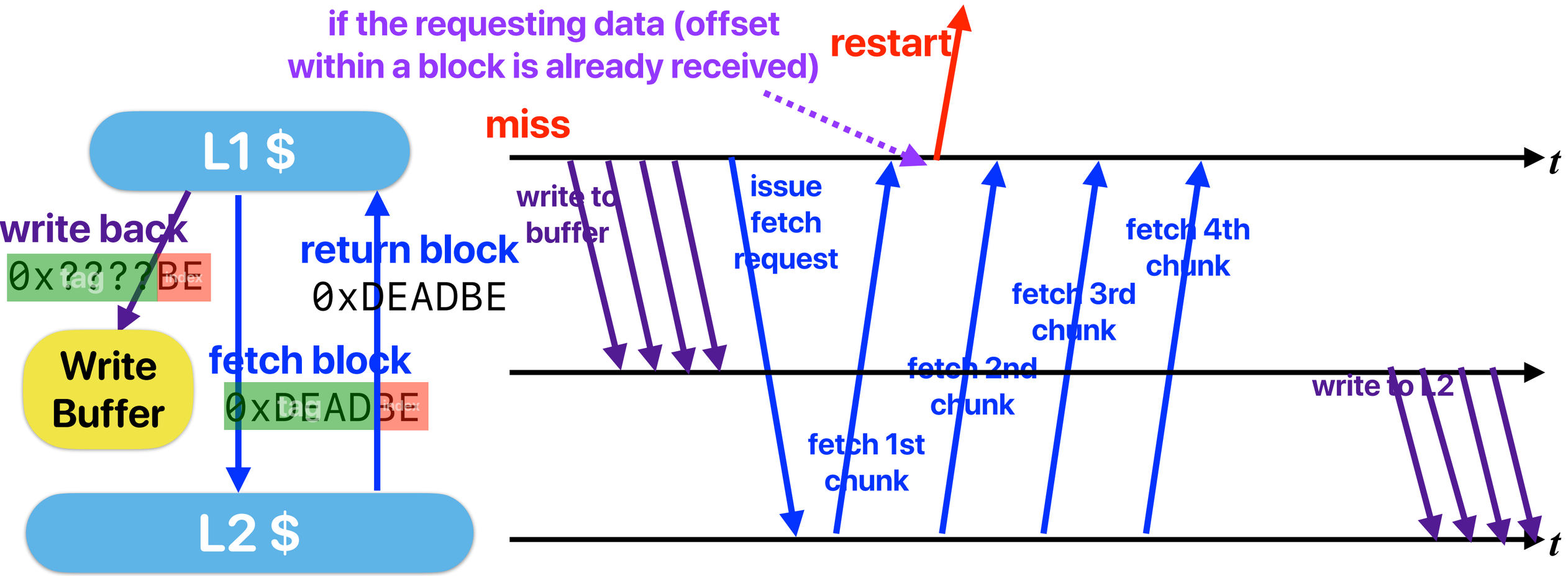
- Don’t wait for full block to be loaded before restarting CPU
  - Early restart—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
  - Critical Word First—Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block. Also called wrapped fetch and requested word first
- Most useful with large blocks
- Spatial locality is a problem; often we want the next sequential word soon, so not always a benefit (early restart).

Can we avoid the overhead of writes?



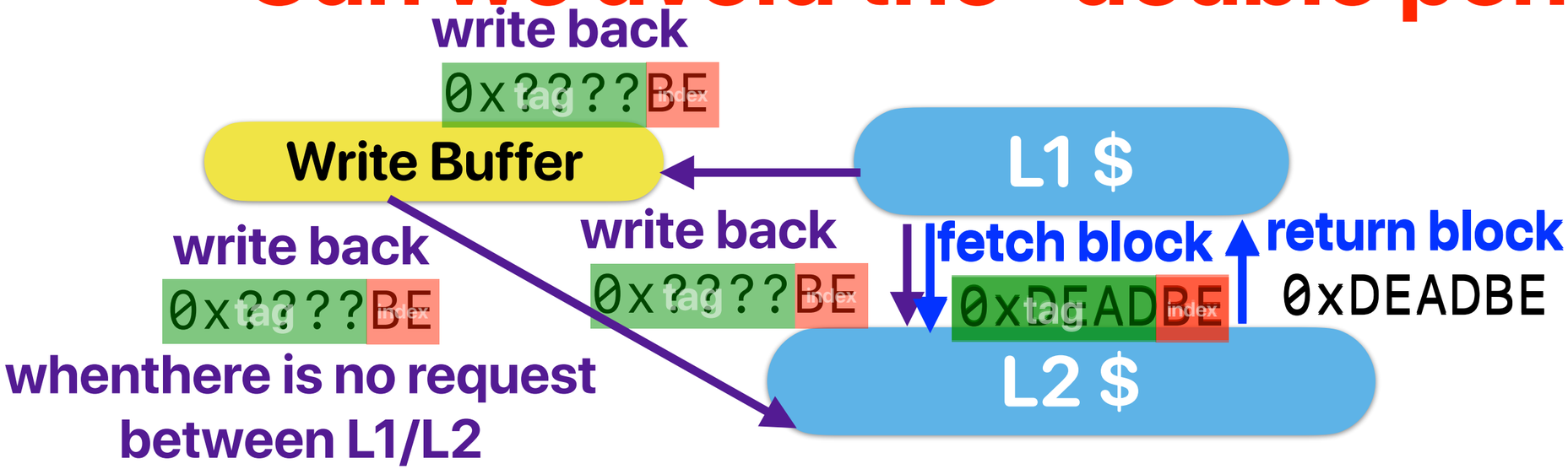
assume the bus between L1/L2 only allows a quarter of the cache block go through it

Write buffer!



assume the bus between L1/L2 only allows a quarter of the cache block go through it

Can we avoid the "double penalty"?



- Every write to lower memory will first write to a small SRAM buffer.
  - store does not incur data hazards, but the pipeline has to stall if the write misses
  - The write buffer will continue writing data to lower-level memory
  - The processor/higher-level memory can response as soon as the data is written to write buffer.
- Write merge
  - Since application has locality, it's highly possible the evicted data have neighboring addresses. Write buffer delays the writes and allows these neighboring data to be grouped together.

Can we avoid the “double penalty”?

- Every write to lower memory will first write to a small SRAM buffer.
  - store does not incur data hazards, but the pipeline has to stall if the write misses
  - The write buffer will continue writing data to lower-level memory
  - The processor/higher-level memory can response as soon as the data is written to write buffer.
- Write merge
  - Since application has locality, it’s highly possible the evicted data have neighboring addresses. Write buffer delays the writes and allows these neighboring data to be grouped together.

Summary of Architectural Optimizations

- Hardware
  - Prefetch — compulsory miss
  - Write buffer — miss penalty
  - Bank/pipeline — miss penalty
  - Critical word first and early restart — miss panelty

How can programmer improve memory performance?

Data structures

Array of structures or structure of arrays

Array of objects

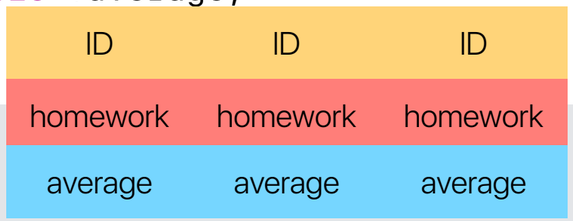
```
struct grades
{
  int id;
  double *homework;
  double average;
};
```

average of each homework

```
for(i=0;i<homework_items; i++)
{
gradesheet[total_number_students].homework[i] = 0.0;
   for(j=0;j<total_number_students;j++)
gradesheet[total_number_students].homework[i]
+=gradesheet[j].homework[i];
   gradesheet[total_number_students].homework[i] /=
(double)total_number_students;
}
```

object of arrays

```
struct grades
{
  int *id;
  double **homework;
  double *average;
};
```




```
for(i = 0;i < homework_items; i++)
{
  gradesheet.homework[i][total_number_students] = 0.0;
  for(j = 0; j <total_number_students;j++)
  {
      gradesheet.homework[i][total_number_students] +=
gradesheet.homework[i][j];
  }
      gradesheet.homework[i][total_number_students] /=
total_number_students;
}
```

Column-store or row-store

- If you're designing an in-memory database system, will you be using

RowId	EmpId	Lastname	Firstname	Salary
1	10	Smith	Joe	40000
2	12	Jones	Mary	50000
3	11	Johnson	Cathy	44000
4	22	Jones	Bob	55000

  - column-store — stores data tables column by column

```
10:001,12:002,11:003,22:004;
Smith:001,Jones:002,Johnson:003,Jones:004;
Joe:001,Mary:002,Cathy:003,Bob:004;
40000:001,50000:002,44000:003,55000:004;
```

if the most frequently used query looks like —

```
select Lastname, Firstname from table
```

  - row-store — stores data tables row by row

```
001:10,Smith,Joe,40000;
002:12,Jones,Mary,50000;
003:11,Johnson,Cathy,44000;
004:22,Jones,Bob,55000;
```

Takeaways: Software Optimizations

- Data layout — capacity miss, conflict miss, compulsory miss

Loop interchange/fission/fusion

Demo — programmer & performance

A

```
for(i = 0; i < ARRAY_SIZE; i++)
{
  for(j = 0; j < ARRAY_SIZE; j++)
  {
    c[i][j] = a[i][j]+b[i][j];
  }
}
```

B

```
for(j = 0; j < ARRAY_SIZE; j++)
{
  for(i = 0; i < ARRAY_SIZE; i++)
  {
    c[i][j] = a[i][j]+b[i][j];
  }
}
```

$O(n^2)$	Complexity	$O(n^2)$
Same	Instruction Count?	Same
Same	Clock Rate	Same
Better	CPI	Worse

Loop optimizations

Loop interchange

A

```
for(i = 0; i < ARRAY_SIZE; i++)
{
  for(j = 0; j < ARRAY_SIZE; j++)
  {
    c[i][j] = a[i][j]+b[i][j];
  }
}
```

B

```
for(j = 0; j < ARRAY_SIZE; j++)
{
  for(i = 0; i < ARRAY_SIZE; i++)
  {
    c[i][j] = a[i][j]+b[i][j];
  }
}
```

Takeaways: Software Optimizations

- Data layout — capacity miss, conflict miss, compulsory miss
- Loop interchange — conflict/capacity miss

What if the code look like this?

- D-L1 Cache configuration of NVIDIA Tegra Orin
  - Size 64KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i]; //load a, b, c and then store to e
for(i = 0; i < 512; i++)
    e[i] /= d[i]; //load e, load d, and then store to e
```

What's the data cache miss rate for this code?

A. ~10%
B. ~20%
C. ~40%
D. ~80%
E. 100%

Loop optimizations

Loop interchange

A

```
for(i = 0; i < ARRAY_SIZE; i++)
{
  for(j = 0; j < ARRAY_SIZE; j++)
  {
    c[i][j] = a[i][j]+b[i][j];
  }
}
```

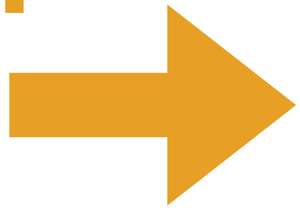
B

```
for(j = 0; j < ARRAY_SIZE; j++)
{
  for(i = 0; i < ARRAY_SIZE; i++)
  {
    c[i][j] = a[i][j]+b[i][j];
  }
}
```

Loop fission

B

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
}
```



A

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i];
for(i = 0; i < 512; i++)
    e[i] /= d[i];
```

Takeaways: Software Optimizations

- Data layout — capacity miss, conflict miss, compulsory miss
- Loop interchange — conflict/capacity miss
- Loop fission — conflict miss — when $ has limited way associativity

Loop optimizations

Loop interchange

A

```
for(i = 0; i < ARRAY_SIZE; i++)
{
  for(j = 0; j < ARRAY_SIZE; j++)
  {
    c[i][j] = a[i][j]+b[i][j];
  }
}
```

B

```
for(j = 0; j < ARRAY_SIZE; j++)
{
  for(i = 0; i < ARRAY_SIZE; i++)
  {
    c[i][j] = a[i][j]+b[i][j];
  }
}
```

Loop fission

B

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
}
```

A

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i];
for(i = 0; i < 512; i++)
    e[i] /= d[i];
```

Loop fusion

A

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++)
    e[i] = a[i] * b[i] + c[i];
for(i = 0; i < 512; i++)
    e[i] /= d[i];
```

B

```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
}
```

Takeaways: Software Optimizations

- Data layout — capacity miss, conflict miss, compulsory miss
- Loop interchange — conflict/capacity miss
- Loop fission — conflict miss — when $ has limited way associativity
- Loop fusion — capacity miss — when $ has enough way associativity

Outline

- Case study: matrix multiplications
- Optimizations for matrix multiplications

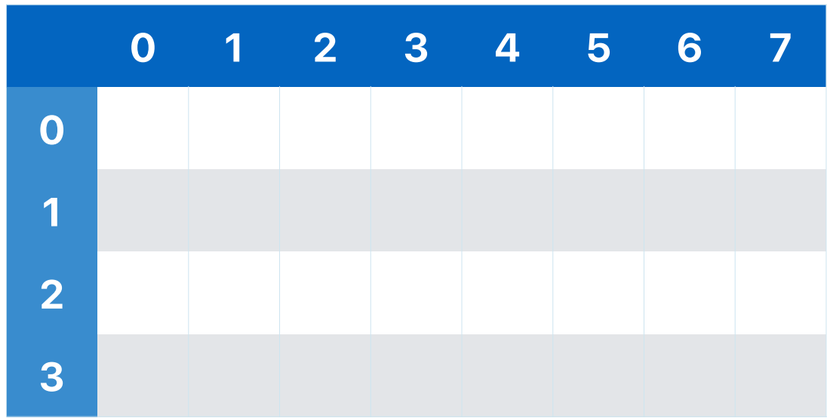
Case study: matrix multiplications

What is an M by N "2-D" array in C?

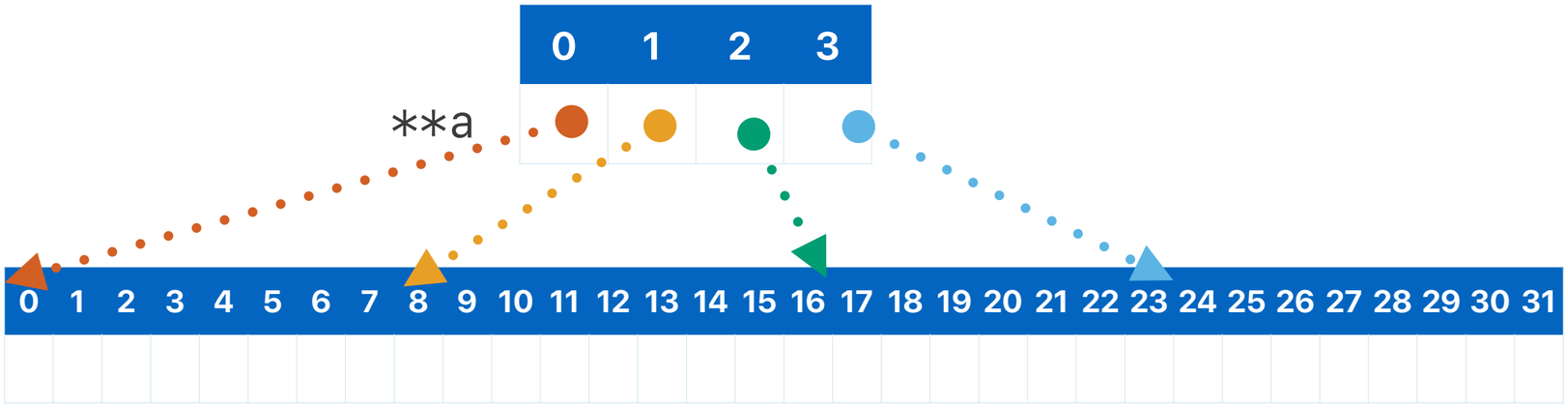
```
a = (double **)malloc(M*sizeof(double *));
for(i = 0; i < N; i++)
{
  a[i] = (double *)malloc(N*sizeof(double));
}
```

a[i][j] is essentially a[i*N+j]

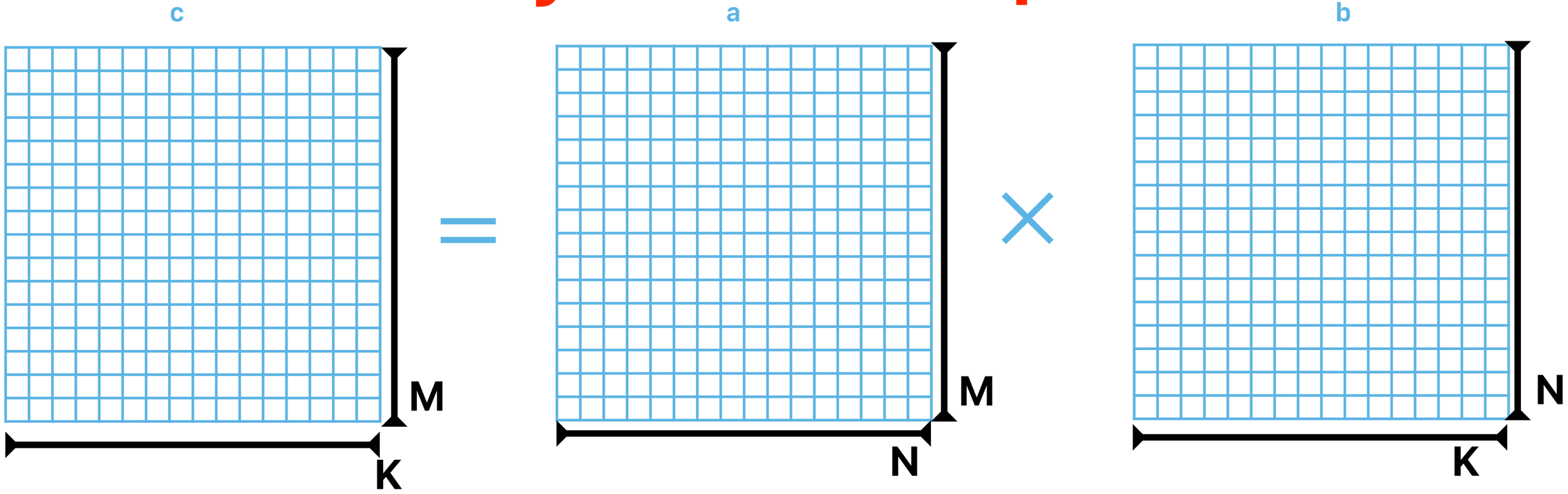
abstraction



physical implementation



Case Study: Matrix Multiplications



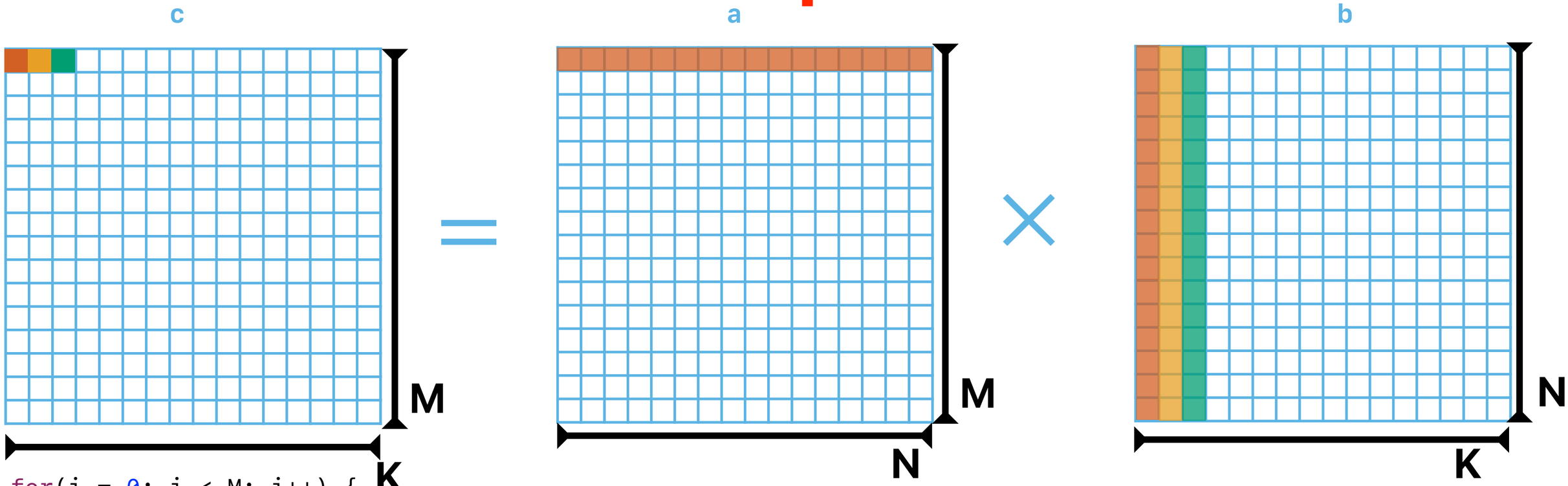

```
for(i = 0; i < M; i++) {
  for(j = 0; j < K; j++) {
    for(k = 0; k < N; k++) {
      c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

Algorithm class tells you it's $O(n^3)$

If M=N=K=1024, it takes about 2 sec

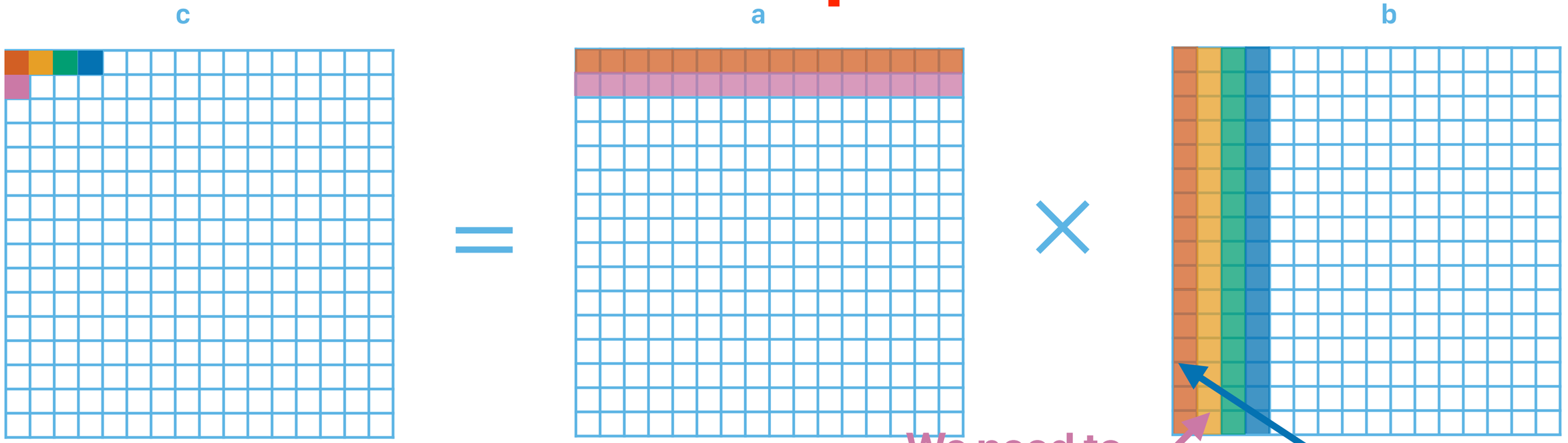
How long is it take when M=N=K=2048?

Matrix Multiplications




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

Matrix Multiplications



- If each dimension of your matrix is 2048
  - Each row takes 2048×8 Bytes = 16 KB
  - Each column takes 2048×8 Bytes = 16 KB
  - The L1-$ of intel Core i7 is 48 KB
  - You can only hold at most 3 rows or 3 columns at most of each matrix!

We need to fetch everything again — capacity miss!

Unlikely to be kept in the cache

Ideas regarding reducing misses in matrix multiplications

- Reducing capacity misses — we need to reduce the length of a row that we visit within a period of time

Mathematical view of MM

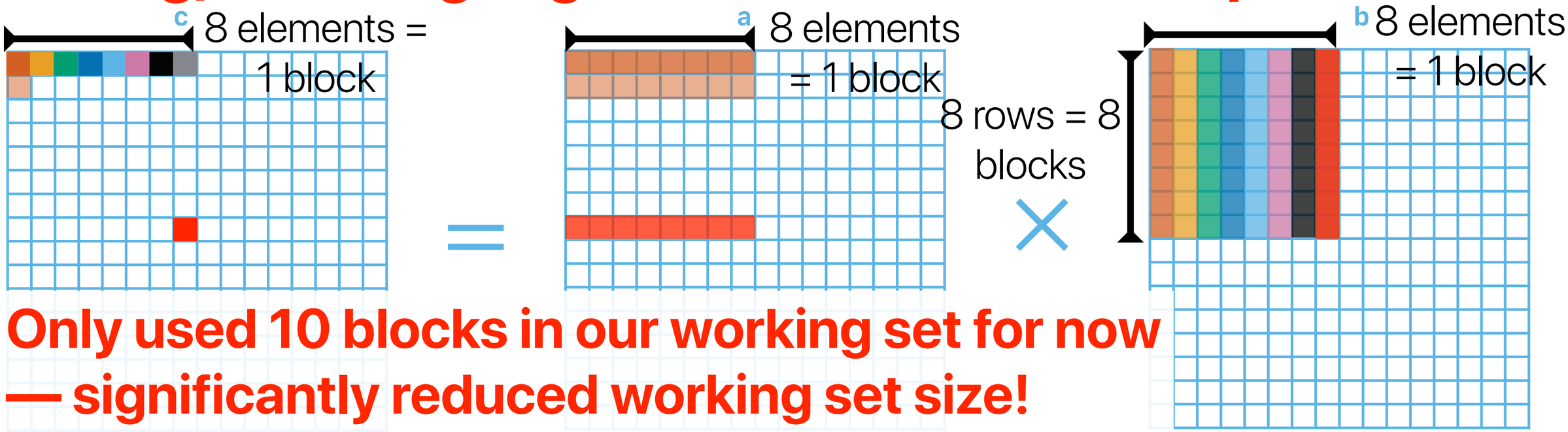
$$c_{i,j} = \sum_{k=0}^{k=N-1} a_{i,k} \times b_{k,j} = \sum_{k=0}^{k=\frac{N}{2}-1} a_{i,k} \times b_{k,j} + \sum_{k=\frac{N}{2}}^{k=N-1} a_{i,k} \times b_{k,j}$$

$$= \sum_{k=0}^{k=\frac{N}{4}-1} a_{i,k} \times b_{k,j} + \sum_{k=\frac{N}{4}}^{k=\frac{N}{2}-1} a_{i,k} \times b_{k,j} + \sum_{k=\frac{N}{2}}^{k=\frac{3N}{4}-1} a_{i,k} \times b_{k,j} + \sum_{k=3N4-1}^{k=N-1} a_{i,k} \times b_{k,j}$$

$$\vdots$$

Let's break up the multiplications and accumulations into something fits in the cache well

Tiling/Blocking Algorithm for Matrix Multiplications



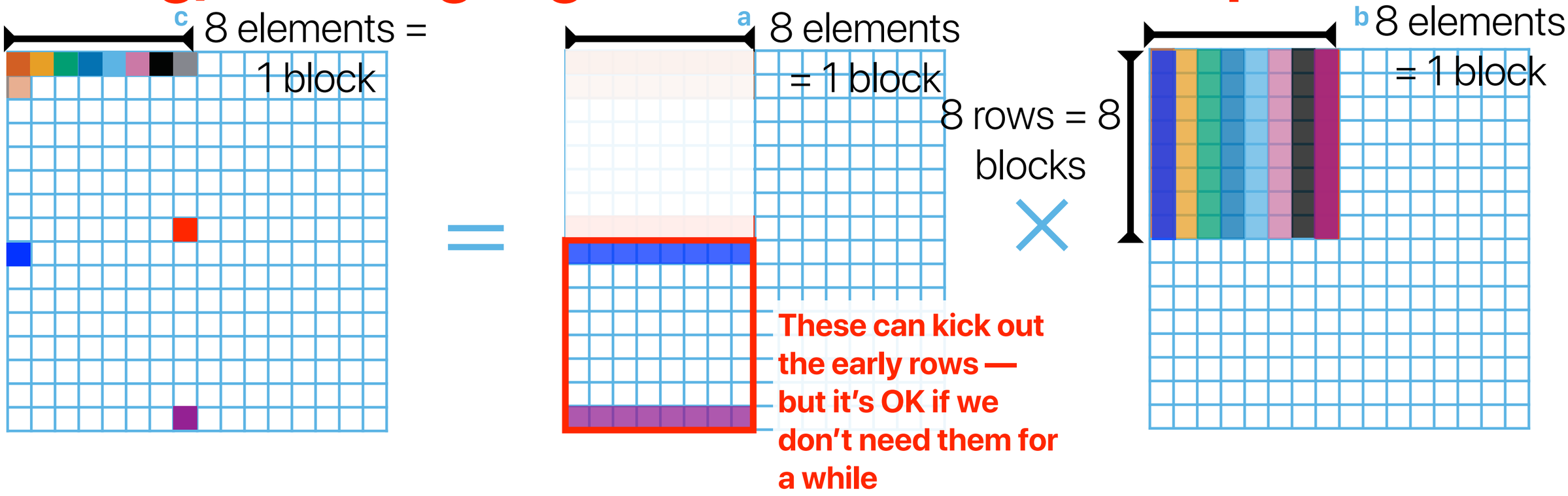
Only used 10 blocks in our working set for now — significantly reduced working set size!

These are still around when we move to the next row in the "tile"

Only compulsory misses —

$$miss\_rate = \frac{total\ misses}{total\ accesses} = \frac{8+8+8}{3\times 8\times 8\times 8} = 0.015625$$

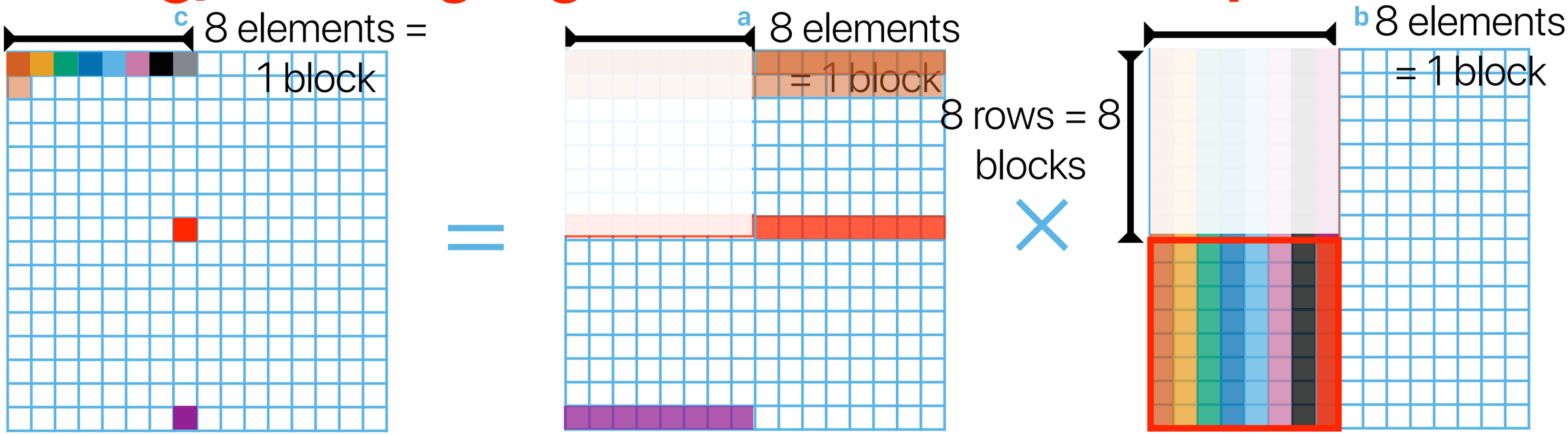
Tiling/Blocking Algorithm for Matrix Multiplications



Bringing miss rate even further lower now —

$$miss\_rate = \frac{total\ misses}{total\ accesses} = \frac{8 + 2 \times 8 + 8}{2 \times 3 \times 8 \times 8 \times 8} = 0.0104$$

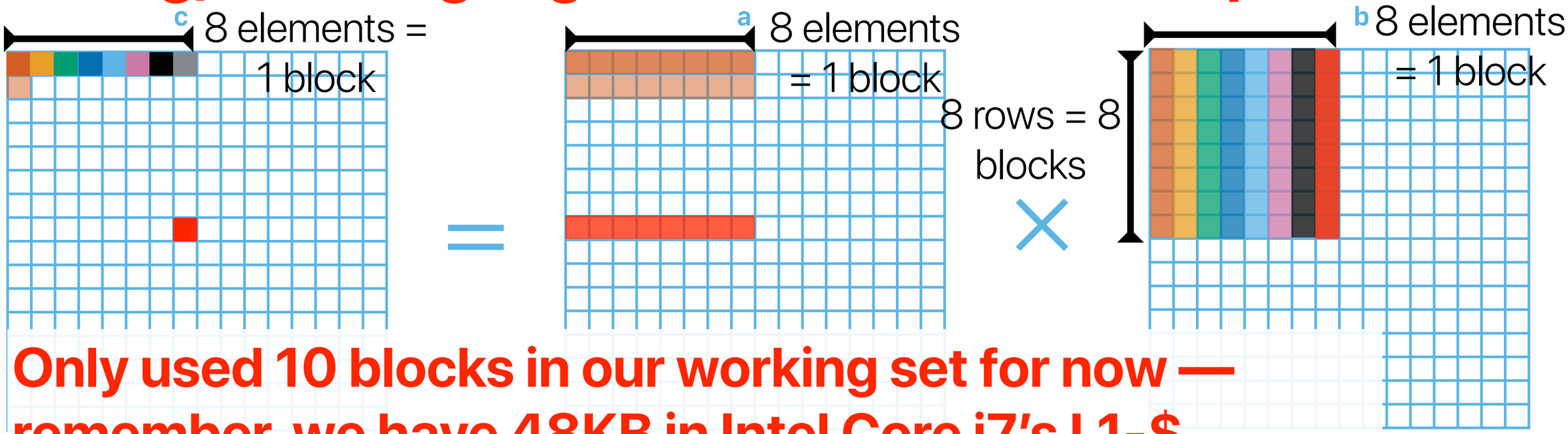
Tiling/Blocking Algorithm for Matrix Multiplications




```
for(i = 0; i < M; i+=tile_size)
    for(j = 0; j < K; j+=tile_size)
        for(k = 0; k < N; k+=tile_size)
            for(ii = i; ii < i+tile_size; ii++)
                for(jj = j; jj < j+tile_size; jj++)
                    for(kk = k; kk < k+tile_size; kk++)
                        c[ii][jj] += a[ii][kk]*b[kk][jj];
```

These can kick out the upper portion of the columns — but it's OK if we don't need them for a while

Tiling/Blocking Algorithm for Matrix Multiplications



Only used 10 blocks in our working set for now — remember, we have 48KB in Intel Core i7's L1-$

We can have at most $\frac{48KB}{3} = 16KB$ **for each matrix "tile"**

So, let's pick tile size to be 32 as $\sqrt{\frac{48KB}{3 \times 8bytes}} \approx 45$

Matrix Multiplication — let's consider "b"

```
for(ii = i; ii < i+tile_size; ii++)
    for(jj = j; jj < j+tile_size; jj++)
        for(kk = k; kk < k+tile_size; kk++)
            c[ii][jj] += a[ii][kk]*b[kk][jj];
```

	Address	Tag	Index
b[0][0]	0x20000	0x20	0x0
b[1][0]	0x24000	0x24	0x0
b[2][0]	0x28000	0x28	0x0
b[3][0]	0x2C000	0x2C	0x0
b[4][0]	0x30000	0x30	0x0
b[5][0]	0x34000	0x34	0x0
b[6][0]	0x38000	0x38	0x0
b[7][0]	0x3C000	0x3C	0x0
b[8][0]	0x40000	0x40	0x0
b[9][0]	0x44000	0x44	0x0
b[10][0]	0x48000	0x48	0x0
b[11][0]	0x4C000	0x4C	0x0
b[12][0]	0x50000	0x50	0x0
b[13][0]	0x54000	0x54	0x0
b[14][0]	0x58000	0x58	0x0
b[15][0]	0x5C000	0x5C	0x0
b[16][0]	0x60000	0x60	0x0

- If the row dimension (N) of your matrix is 2048, each row element with the same column index is

$$2048 \times 8 = 16384 = 0x4000$$

away from each other

Each set can store only 12 blocks! So we will start to kick out b[0][0-7], b[1][0-7] …

Now, when we work on c[0][1]

	Address	Tag	Index
b[0][0]	0x20000	0x20	0x0
b[1][0]	0x24000	0x24	0x0
b[2][0]	0x28000	0x28	0x0
b[3][0]	0x2C000	0x2C	0x0
b[4][0]	0x30000	0x30	0x0
b[5][0]	0x34000	0x34	0x0
b[6][0]	0x38000	0x38	0x0
b[7][0]	0x3C000	0x3C	0x0
b[8][0]	0x40000	0x40	0x0
b[9][0]	0x44000	0x44	0x0
b[10][0]	0x48000	0x48	0x0
b[11][0]	0x4C000	0x4C	0x0
b[12][0]	0x50000	0x50	0x0
b[13][0]	0x54000	0x54	0x0
b[14][0]	0x58000	0x58	0x0
b[15][0]	0x5C000	0x5C	0x0
b[16][0]	0x60000	0x60	0x0

	Address	Tag	Index	
b[0][1]	0x20008	0x20	0x0	Conflict Miss
b[1][1]	0x24008	0x24	0x0	Conflict Miss
b[2][1]	0x28008	0x28	0x0	Conflict Miss
b[3][1]	0x2C008	0x2C	0x0	Conflict Miss
b[4][1]	0x30008	0x30	0x0	Conflict Miss
b[5][1]	0x34008	0x34	0x0	Conflict Miss
b[6][1]	0x38008	0x38	0x0	Conflict Miss
b[7][1]	0x3C008	0x3C	0x0	Conflict Miss
b[8][1]	0x40008	0x40	0x0	Conflict Miss
b[9][1]	0x44008	0x44	0x0	Conflict Miss
b[10][1]	0x48008	0x48	0x0	Conflict Miss
b[11][1]	0x4C008	0x4C	0x0	Conflict Miss
b[12][1]	0x50008	0x50	0x0	Conflict Miss
b[13][1]	0x54008	0x54	0x0	Conflict Miss
b[14][1]	0x58008	0x58	0x0	Conflict Miss
b[15][1]	0x5C008	0x5C	0x0	Conflict Miss
b[16][1]	0x60008	0x60	0x0	Conflict Miss

Each set can store only 12 blocks! So we will start to kick out b[0][0-7], b[1][0-7] ...

Matrix Multiplication — let's consider "b"

```
for(ii = i; ii < i+tile_size; ii++)
    for(jj = j; jj < j+tile_size; jj++)
        for(kk = k; kk < k+tile_size; kk++)
            c[ii][jj] += a[ii][kk]*b[kk][jj];
```

	Address	Tag	Index
b[0][0]	0x20000	0x20	0x0
b[1][0]	0x24000	0x24	0x0
b[2][0]	0x28000	0x28	0x0
b[3][0]	0x2C000	0x2C	0x0
b[4][0]	0x30000	0x30	0x0
b[5][0]	0x34000	0x34	0x0
b[6][0]	0x38000	0x38	0x0
b[7][0]	0x3C000	0x3C	0x0
b[8][0]	0x40000	0x40	0x0
b[9][0]	0x44000	0x44	0x0
b[10][0]	0x48000	0x48	0x0
b[11][0]	0x4C000	0x4C	0x0
b[12][0]	0x50000	0x50	0x0
b[13][0]	0x54000	0x54	0x0
b[14][0]	0x58000	0x58	0x0
b[15][0]	0x5C000	0x5C	0x0
b[16][0]	0x60000	0x60	0x0

- If the row dimension of your matrix is 2048, each row element with the same column index is $2048 \times 8 = 16384 = 0x4000$ away from each other

If we stop at somewhere before 12 blocks, we should be fine!

Since each block has 8 elements, let's break up in 8 for now

— 8 elements from a[i]

— 8 columns each covers 8 rows

Ideas regarding reducing misses in matrix multiplications

- Reducing capacity misses — we need to reduce the length of a row that we visit within a period of time
- Reducing conflict misses — we need to ensure an appropriate tile size would not lead to conflict in sets

Use registers wisely

```
for(i = 0; i < M; i+=tile_size)
    for(j = 0; j < K; j+=tile_size)
        for(k = 0; k < N; k+=tile_size)
            for(ii = i; ii < i+tile_size; ii++)
                for(jj = j; jj < j+tile_size; jj++)
                    for(kk = k; kk < k+tile_size; kk++)
                        c[ii][jj] += a[ii][kk]*b_t[jj][kk];
```

This will create a memory access!

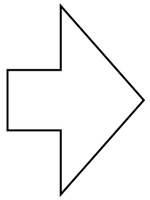
```
for(i = 0; i < M; i+=tile_size)
    for(j = 0; j < K; j+=tile_size)
        for(k = 0; k < N; k+=tile_size)
            for(ii = i; ii < i+tile_size; ii++)
                for(jj = j; jj < j+tile_size; jj++)
                {
                    result = 0;
                    for(kk = k; kk < k+tile_size; kk++)
                        result += a[ii][kk]*b_t[jj][kk];
                    c[ii][jj] += result;
                }
```

The compiler will try to make result in a register

— without writing code in this way, compiler may not optimize

Matrix Transpose

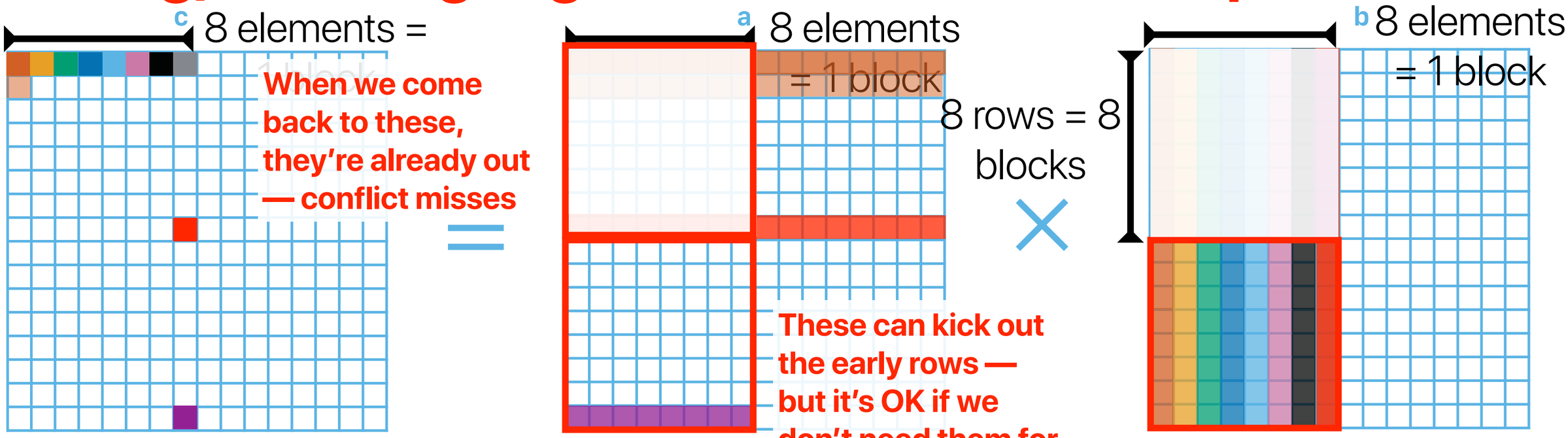
```
for(i = 0; i < M; i+=tile_size) {
  for(j = 0; j < N; j+=tile_size) {
    for(k = 0; k < K; k+=tile_size) {
      for(ii = i; ii < i+tile_size; ii++)
        for(jj = j; jj < j+tile_size; jj++)
          for(kk = k; kk < k+tile_size; kk++)
            c[ii][jj] += a[ii][kk]*b[kk][jj];
    }
  }
}
```



```
// Transpose matrix b into b_t
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
  for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
      b_t[i][j] += b[j][i];
  }
}
```

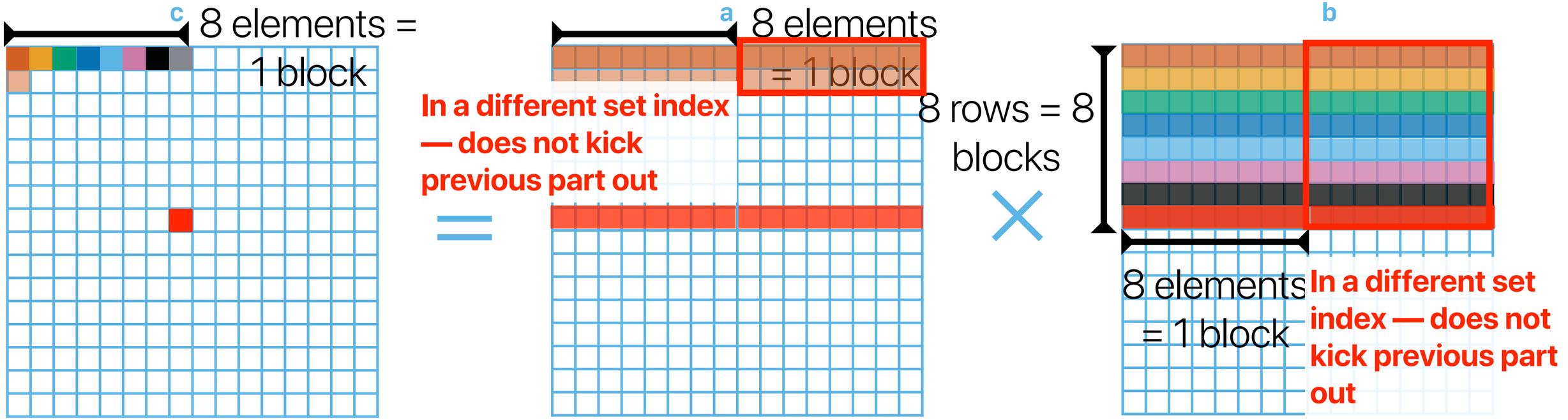
```
for(i = 0; i < M; i+=tile_size) {
  for(j = 0; j < N; j+=tile_size) {
    for(k = 0; k < K; k+=tile_size) {
      for(ii = i; ii < i+tile_size; ii++)
        for(jj = j; jj < j+tile_size; jj++)
          for(kk = k; kk < k+tile_size; kk++)
            // Compute on b_t
            c[ii][jj] += a[ii][kk]*b_t[jj][kk];
    }
  }
}
```

Tiling/Blocking Algorithm for Matrix Multiplications




```
for(i = 0; i < M; i+=tile_size)
    for(j = 0; j < K; j+=tile_size)
        for(k = 0; k < N; k+=tile_size)
            for(ii = i; ii < i+tile_size; ii++)
                for(jj = j; jj < j+tile_size; jj++)
                    for(kk = k; kk < k+tile_size; kk++)
                        c[ii][jj] += a[ii][kk]*b[kk][jj];
```

Tiling/Blocking Algorithm for Transposed Matrix Multiplications



We can make the "tile_size" larger without interfacing conflict misses

```
for(i = 0; i < M; i+=tile_size)
    for(j = 0; j < K; j+=tile_size)
        for(k = 0; k < N; k+=tile_size)
            for(ii = i; ii < i+tile_size; ii++)
                for(jj = j; jj < j+tile_size; jj++)
                    for(kk = k; kk < k+tile_size; kk++)
                        c[ii][jj] += a[ii][kk]*b_t[jj][kk];
```

The effect of transposition

Block/tile

size	tile_size	IC	Cycles	CPI	CT_ns	ET_s	DL1_miss_rate
2048	4	97766571061	23972375695	0.245200	0.193290	4.633619	0.015394
2048	8	81047436195	21583826614	0.266311	0.193122	4.168303	0.010260
2048	16	74472586117	19268082018	0.258727	0.193325	3.724997	0.071558
2048	32	71543547491	27661109860	0.386633	0.193218	5.344616	0.217189
2048	64	70151970961	32605985592	0.464791	[illegible]	6.301039	0.242202
2048	128	69470212062	34530336995	[illegible]	[illegible]	6.672484	0.246013
2048	256	69131368754	35151111975	0.508468	0.193311	6.795085	0.246800

Best performing from 16 to 64?

Block + Transpose

size	tile_size	IC	Cycles	CPI	CT_ns	ET_s	DL1_miss_rate
2048	8	70351352368	16009523067	0.227565	0.193097	3.091384	0.001897
2048	16	64810353582	15145593176	0.233691	0.193199	2.926121	0.026059
2048	32	62397963236	14854143892	0.238055	0.193161	2.869243	0.040979
2048	64	[illegible]	[illegible]	0.223012	0.193260	2.640043	0.023464
2048	128	60710004318	[illegible]	0.275478	0.193123	3.229842	0.018013
2048	256	60438882203	17003851823	0.281340	0.193330	3.287351	0.012972

Transpose improves the miss rate at larger tile size!

The effect of transposition + register

Block + Transpose

size	tile_size	IC	Cycles	CPI	CT_ns	ET_s	DL1_miss_rate	DL1_accesses
2048	8	70351352368	16009523067	0.227565	0.193097	3.091384	0.001897	28769906635
2048	16	64810353582	15145593176	0.233691	0.193199	2.926121	0.026059	27045466371
2048	32	62397963236	14854143892	0.238055	0.193161	2.869243	0.040979	26365975027
2048	64	61255133491	13660607640	0.223012	0.193260	2.640043	0.023464	26061062917
2048	128	60710004318	16724248174	0.275478	0.193123	3.229842	0.018013	25921112408
2048	256	60438882203	17003851823	0.281340	0.193330	3.287351	0.012972	25852375287

Significant reduction of memory accesses

Block + Transpose + Reg.

size	tile_size	IC	Cycles	CPI	CT_ns	ET_s	DL1_miss_rate	DL1_accesses
2048	8	60667686406	11604601609	0.191281	0.193985	2.251117	0.003363	15849319789
2048	16	49205927530	10032628049	0.203891	0.193373	1.940040	0.066542	11986874817
2048	32	43854347828	9736190510	0.222012	0.193008	1.879164	0.096360	10247265803
2048	64	41246516681	10038942768	0.243389	0.193141	1.938936	0.069853	9413044702
2048	128	39962465083	[illegible]	[illegible]	[illegible]	2.299423	0.045847	9005357520
2048	256	39326412762	14051616706	0.357307	0.193021	2.712263	0.032001	8804127506

Best performing at 32

Tiles do not have to be “squares”

```
for(i = 0; i < M; i+=tile_size_y) {
    for(j = 0; j < K; j+=tile_size_y) {
      for(k = 0; k < N; k+=tile_size_x) {
          for(ii = i; ii < i+tile_size_y; ii++)
              for(jj = j; jj < j+tile_size_y; jj++) {
                  result = 0;
                  for(kk = k; kk < k+tile_size_x; kk++)
                      result += a[ii][kk]*b[jj][kk];
                  c[ii][jj] += result;
              }
          }
      }
   }
}
```

If we could have a rectangular tile + transposition

Block + Transpose + Reg.

size	tile_size	IC	Cycles	CPI	CT_ns	ET_s	DL1_miss_rate
2048	8	60667686406	11604601609	0.191281	0.193985	2.251117	0.003363
2048	16	49205927530	10032628049	0.203891	0.193373	1.940040	0.066542
2048	32	43854347828	9736190510	0.222012	0.193008	1.879164	0.096360
2048	64	41246516681	10038942768	0.243389	0.193141	1.938936	0.069853
2048	128	39962465083	11887280358	0.297461	0.193436	2.299423	0.045847
2048	256	393[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0.032001

Very low miss rate compared to 32x32

Rect. Block + Transpose + Reg.

size	tile_size_x	tile_size_y	IC	Cycles	CPI	CT_ns	ET_s	DL1_miss_rate
2048	8	8	60696024519	11624419280	0.191519	0.193479	2.249077	0.003664
2048	8	16	59757245622	12523769669	0.209577	0.193096	2.418293	0.022092
2048	16	8	49689499294	11415401958	0.229735	0.193200	2.205459	0.003957
2048	16	16	[illegible]	[illegible]	[illegible]	[illegible]	1.935619	0.067519
2048	32	16	[illegible]	[illegible]	[illegible]	[illegible]	1.879111	0.066730
2048	32	8	44182255983	9353772552	0.211709	0.193233	1.807453	0.005696
2048	64	8	41431394002	9636426904	0.232588	0.193230	1.862043	0.005849
2048	128	8	40059698391	11599727168	0.289561	0.193133	2.240289	0.006533
2048	256	8	39376445298	13785740954	0.350101	0.193180	2.663130	0.004190

Best performing at 32x8

Why is “8” not the best performing?

size	tile_size	IC	Cycles	CPI	CT_ns	ET_s	DL1_miss_rate	DL1_accesses
2048	4	97765686275	24149510064	0.247014	0.193189	4.665430	0.015102	43641501149
2048	8	80996985555	21043742544	0.259809	0.193444	4.070776	0.010135	38128531445
2048	16	74473114435	19369857501	0.260092	0.193204	3.742332	0.071790	36105122733
2048	32	71543334296	27812871208	0.388756	0.193112	5.371009	0.217011	35214370198

More instructions due to more loop control overhead!

Why 1% and 7% miss rate do not make significant difference?

Why is “8” not the best performing?

size	tile_size	IC	Cycles	CPI	CT_ns	ET_s	DL1_miss_rate
2048	4	97766571061	23972375695	0.245200	0.193290	4.633619	0.015394
2048	8	81047436195	21583826614	0.266311	0.193122	4.168303	0.010260
2048	16	74472586117	19268082018	0.258727	0.193325	3.724997	0.071558
2048	32	71543547491	27661109860	0.386633	0.193218	5.344616	0.217189
2048	64	70151970961	32605985592	0.464791	0.193248	6.301039	0.242202
2048	128	69470212062	34530336995	0.497052	0.193235	6.672484	0.246013
2048	256	69131368754	35151111975	0.508468	0.193311	6.795085	0.246800
2048	512	68985162572	47048159619	0.682004	0.193298	9.094299	0.239775

More instructions due to more loop control overhead!

Best performing at 16?

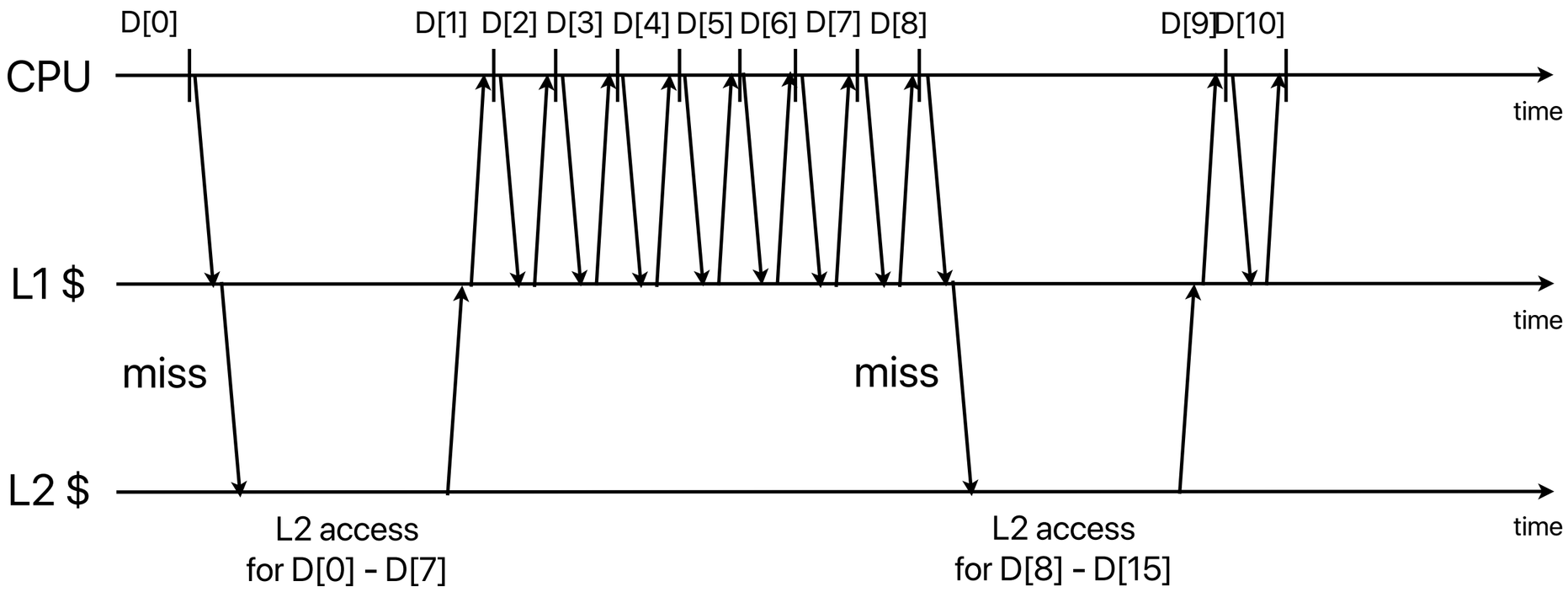
“8” indeed has the best miss rate — and matches our predictions!

Takeaways: Software Optimizations

- Data layout — capacity miss, conflict miss, compulsory miss
- Loop interchange — conflict/capacity miss
- Loop fission — conflict miss — when $ has limited way associativity
- Loop fusion — capacity miss — when $ has enough way associativity
- Blocking/tiling — capacity miss, conflict miss
- Matrix transpose (a technique changes layout) — conflict misses
- Using registers whenever possible — reduce memory accesses!

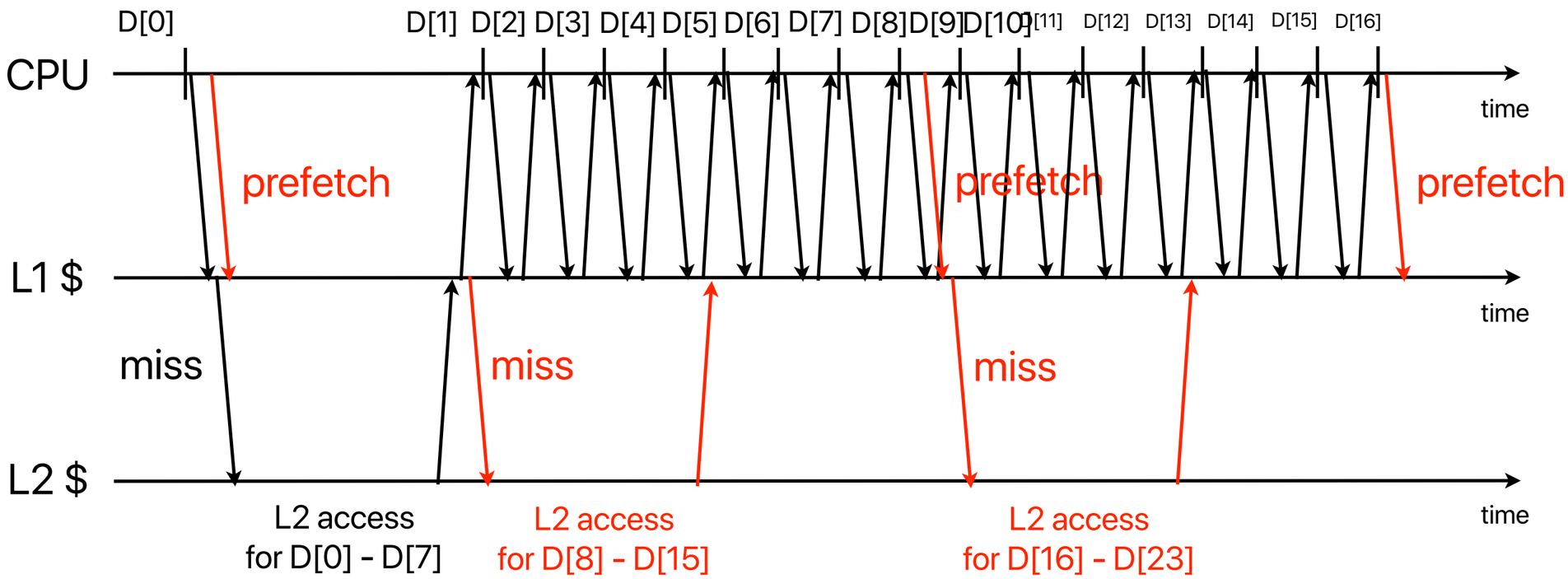
Characteristic of memory accesses

```
for(i = 0;i < 1000000; i++) {
     D[i] = rand();
}
```



Prefetching

```
for(i = 0;i < 1000000; i++) {
      D[i] = rand();
      // prefetch D[i+8] if i % 8 == 0
}
```



Software Prefetching — through prefetching instructions

- x86 provide prefetch instructions
- As a programmer, you may insert `_mm_prefetch` in x86 programs to perform software prefetch for your code
- gcc also has a flag "-fprefetch-loop-arrays" to automatically insert software prefetch instructions

Takeaways: Software Optimizations

- Data layout — capacity miss, conflict miss, compulsory miss
- Loop interchange — conflict/capacity miss
- Loop fission — conflict miss — when $ has limited way associativity
- Loop fusion — capacity miss — when $ has enough way associativity
- Blocking/tiling — capacity miss, conflict miss
- Matrix transpose (a technique changes layout) — conflict misses
- Using registers whenever possible — reduce memory accesses!

Software Prefetching — through prefetching instructions

- x86 provide prefetch instructions
- As a programmer, you may insert `_mm_prefetch` in x86 programs to perform software prefetch for your code
- gcc also has a flag "-fprefetch-loop-arrays" to automatically insert software prefetch instructions

Takeaways: Software Optimizations

- Data layout — capacity miss, conflict miss, compulsory miss
- Loop interchange — conflict/capacity miss
- Loop fission — conflict miss — when $ has limited way associativity
- Loop fusion — capacity miss — when $ has enough way associativity
- Blocking/tiling — capacity miss, conflict miss