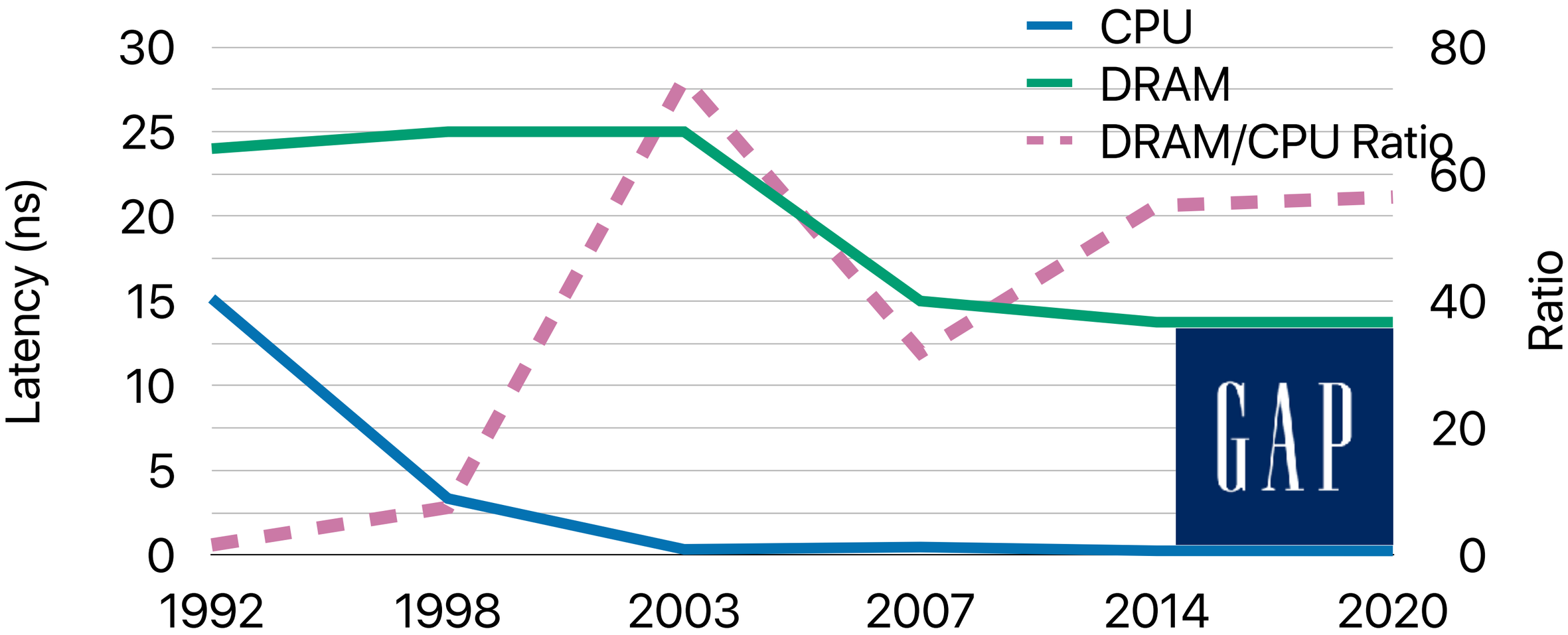
Memory Hierarchy (3): Cache misses and their remedies — the hardware version

Hung-Wei Tseng

Recap: 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

Recap: Memory Hierarchy

fastest

< 1ns

a few ns

tens of ns

us/ms

Processor

Processor Core

Registers

SRAM $

DRAM

Storage

TBs

larger

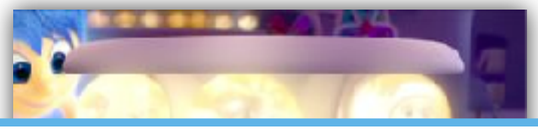
fastest

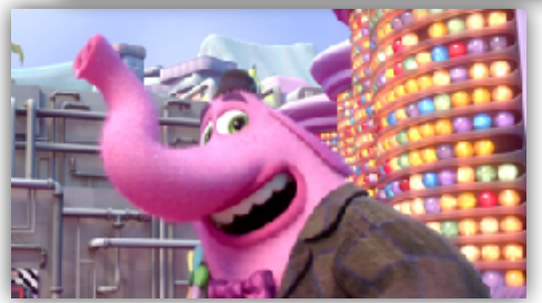
L1 $

L2 $

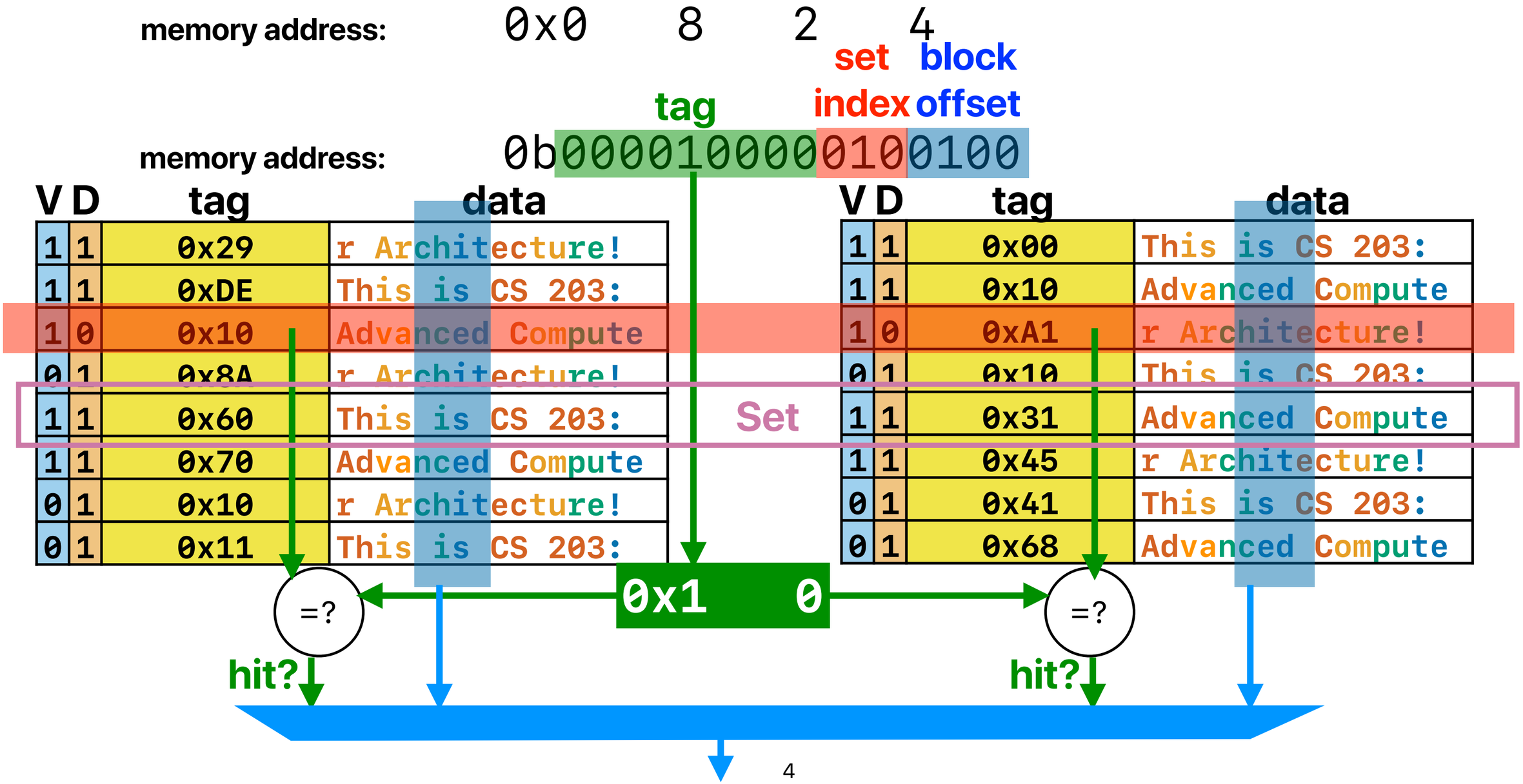
L3 $

larger

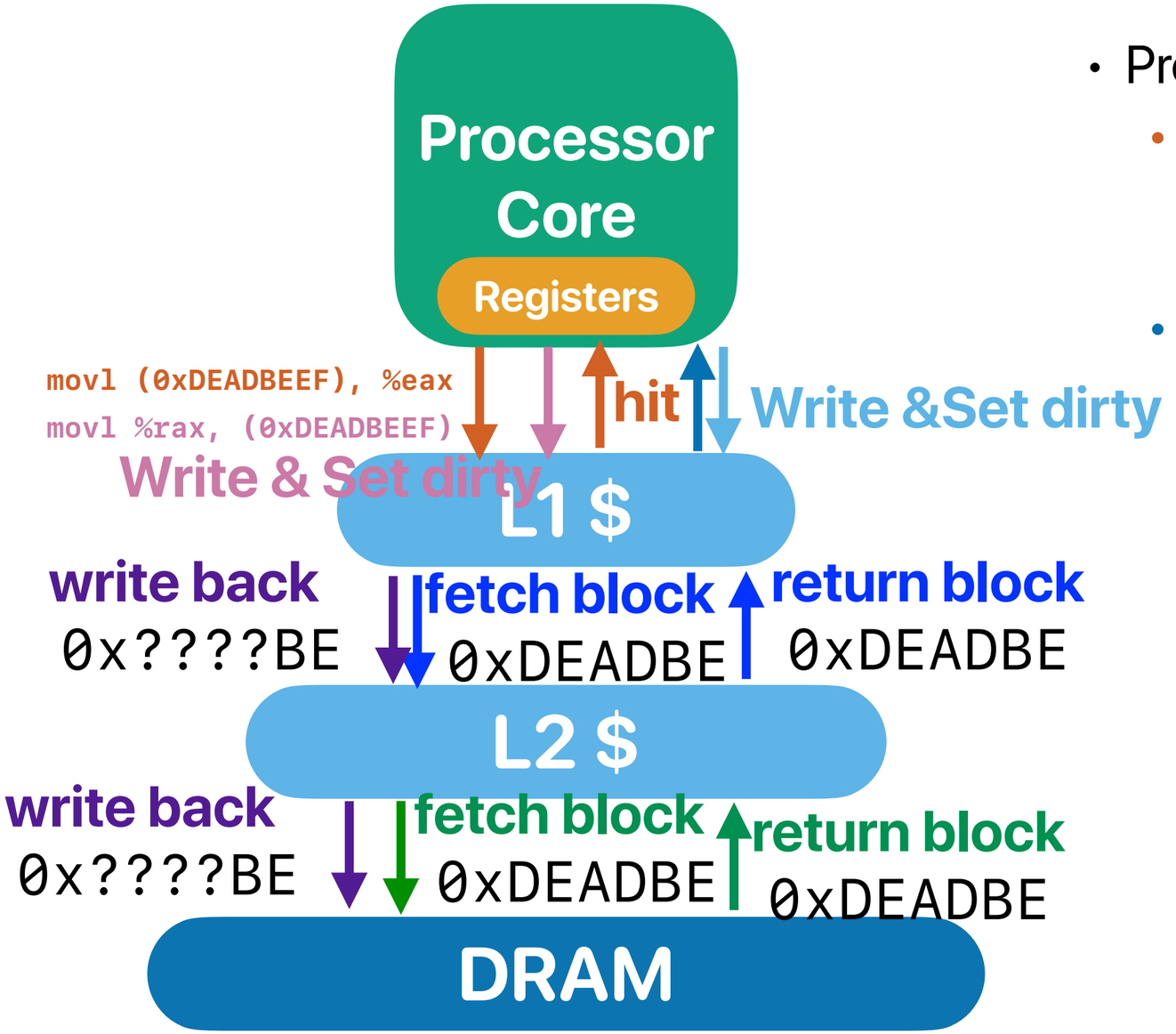




Recap: Way-associative cache



Full picture: Processor/cache interaction

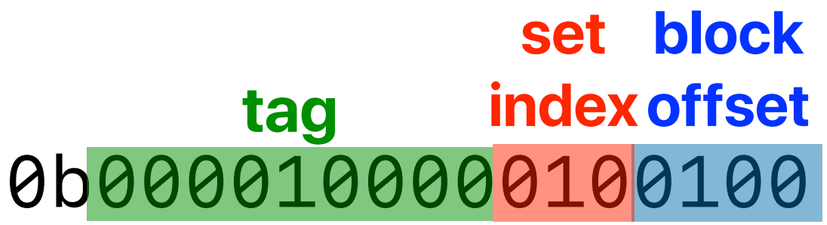


- Processor sends memory access request to L1-$
  - **if hit & it's a read**
    - **Read: return data**
    - **Write: Update "ONLY" in L1 and set DIRTY**
  - **if miss**
    - If there an empty block — place the data there
    - If NOT (most frequent case) — select a **victim block**
      - Least 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**

Review: 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 (Cacheline)
  - How many bytes in a block
- **S**: Number of **S**ets:
  - A set contains blocks sharing the same index
  - 1 for fully associate cache
- 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 64 bits for 64-bit machine
- $\frac{address}{block\_size} \pmod{S}$ = set index

memory address:



Evaluating the cache miss rate of the code

- Understanding the geometry (i.e., C = A B S) of your cache
- List a sequence of memory addresses of data accesses
- Partition each address based on the geometry of the cache
- Simulate the cache to see if it’s a hit or a miss, also, what kind of misses

NVIDIA Tegra X1

100% miss rate!

- Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 8192; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

C = ABS
32KB = 4 * 64 * S
S = 128
offset = lg(64) = 6 bits
index = lg(128) = 7 bits
tag = the rest bits

tag index offset

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[0]	0x10000	0b0001000000000000000000	0x8	0x0	Miss	
b[0]	0x20000	0b0010000000000000000000	0x10	0x0	Miss	
c[0]	0x30000	0b0011000000000000000000	0x18	0x0	Miss	
d[0]	0x40000	0b0100000000000000000000	0x20	0x0	Miss	
e[0]	0x50000	0b0101000000000000000000	0x28	0x0	Miss	a[0–7]
a[1]	0x10008	0b0001000000000000001000	0x8	0x0	Miss	b[0–7]
b[1]	0x20008	0b0010000000000000001000	0x10	0x0	Miss	c[0–7]
c[1]	0x30008	0b0011000000000000001000	0x18	0x0	Miss	d[0–7]
d[1]	0x40008	0b0100000000000000001000	0x20	0x0	Miss	e[0–7]
e[1]	0x50008	0b0101000000000000001000	0x28	0x0	Miss	a[0–7]
⋮	⋮	⋮	⋮	⋮	⋮	⋮

NVIDIA Tegra X1 (cont.)

- Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 8192; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

C = ABS
32KB = 4 * 64 * S
S = 128
offset = lg(64) = 6 bits
index = lg(128) = 7 bits
tag = the rest bits

tag index offset

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[7]	0x10038	0b0001000000000111000	0x8	0x0	Miss	
b[7]	0x20038	0b0010000000000111000	0x10	0x0	Miss	
c[7]	0x30038	0b0011000000000111000	0x18	0x0	Miss	
d[7]	0x40038	0b0100000000000111000	0x20	0x0	Miss	
e[7]	0x50038	0b0101000000000111000	0x28	0x0	Miss	a[0–7]
a[8]	0x10040	0b0001000000001000000	0x8	0x1	Miss	
b[8]	0x20040	0b0010000000001000000	0x10	0x1	Miss	
c[8]	0x30040	0b0011000000001000000	0x18	0x1	Miss	
d[8]	0x40040	0b0100000000001000000	0x20	0x1	Miss	
e[8]	0x50040	0b0101000000001000000	0x28	0x1	Miss	a[8–15]
⋮	⋮	⋮	⋮	⋮	⋮	⋮

100% miss rate!

Outline

- Simulate code behavior on caches (cont.)
- Sources of cache misses
- A, B, C, S and cache misses
- Architectural support for optimizing cache performance

intel Core i7

- D-L1 Cache configuration of intel Core i7
  - Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 8192; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

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

A. 12.5%
B. 56.25%
C. 66.67%
D. 68.75%
E. 100%

intel Core i7

- Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 8192; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

C = ABS
48KB = 12 * 64 * S
S = 64
offset = lg(64) = 6 bits
index = lg(64) = 6 bits
tag = the rest bits

tag index offset

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[0]	0x10000	0b00010000000000000000	0x10	0x0	Miss	
b[0]	0x20000	0b00100000000000000000	0x20	0x0	Miss	
c[0]	0x30000	0b00110000000000000000	0x30	0x0	Miss	
d[0]	0x40000	0b01000000000000000000	0x40	0x0	Miss	
e[0]	0x50000	0b01010000000000000000	0x50	0x0	Miss	
a[1]	0x10008	0b00010000000000001000	0x10	0x0	Hit	
b[1]	0x20008	0b00100000000000001000	0x20	0x0	Hit	
c[1]	0x30008	0b00110000000000001000	0x30	0x0	Hit	
d[1]	0x40008	0b01000000000000001000	0x40	0x0	Hit	
e[1]	0x50008	0b01010000000000001000	0x50	0x0	Hit	
⋮	⋮	⋮	⋮	⋮	⋮	⋮

intel Core i7 (cont.)

- Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 8192; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

C = ABS
48KB = 12 * 64 * S
S = 64
offset = lg(64) = 6 bits
index = lg(64) = 6 bits
tag = the rest bits

tag index offset

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[7]	0x10038	0b0001000000000111000	0x10	0x0	Hit	
b[7]	0x20038	0b0010000000000111000	0x20	0x0	Hit	
c[7]	0x30038	0b0011000000000111000	0x30	0x0	Hit	
d[7]	0x40038	0b0100000000000111000	0x40	0x0	Hit	
e[7]	0x50038	0b0101000000000111000	0x50	0x0	Hit	
a[8]	0x10040	0b0001000000000001000000	0x10	0x1	Miss	
b[8]	0x20040	0b0010000000000001000000	0x20	0x1	Miss	
c[8]	0x30040	0b0011000000000001000000	0x30	0x1	Miss	
d[8]	0x40040	0b0100000000000001000000	0x40	0x1	Miss	
e[8]	0x50040	0b0101000000000001000000	0x50	0x1	Miss	
a[9]	0x10048	0b0001000000000001001000	0x10	0x1	Hit	
b[9]	0x20048	0b0010000000000001001000	0x20	0x1	Hit	
c[9]	0x30048	0b0011000000000001001000	0x30	0x1	Hit	
d[9]	0x40048	0b0100000000000001001000	0x40	0x1	Hit	

$$\frac{5 \times \frac{512}{8}}{5 \times 512} = \frac{1}{8} = 12.5\,\%$$

Miss when the array index is a multiply of 8!

intel Core i7

- D-L1 Cache configuration of intel Core i7
  - Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 8192; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

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

A. 12.5%
B. 56.25%
C. 66.67%
D. 68.75%
E. 100%

Take-aways: cache misses and their remedies

- Our code behaves differently on different cache configurations

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
  - Working set size means the total size of **cache blocks** we visit before the next reuse of the current block
- Conflict miss
  - Required data replaced by block(s) mapping to the same set
  - Similar collision in hash

NVIDIA Tegra X1

- D-L1 Cache configuration of NVIDIA Tegra X1
  - Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 8192; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

How many of the cache misses are **conflict** misses?

A. 12.5%
B. 66.67%
C. 68.75%
D. 87.5%
E. 100%

NVIDIA Tegra X1

100% miss rate!

- Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 8192; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

C = ABS
32KB = 4 * 64 * S
S = 128
offset = lg(64) = 6 bits
index = lg(128) = 7 bits
tag = the rest bits

tag index offset

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[0]	0x10000	0b0001000000000000000000	0x8	0x0	Compulsory Miss	
b[0]	0x20000	0b0010000000000000000000	0x10	0x0	Compulsory Miss	
c[0]	0x30000	0b0011000000000000000000	0x18	0x0	Compulsory Miss	
d[0]	0x40000	0b0100000000000000000000	0x20	0x0	Compulsory Miss	
e[0]	0x50000	0b0101000000000000000000	0x28	0x0	Compulsory Miss	a[0–7]
a[1]	0x10008	0b0001000000000000001000	0x8	0x0	Conflict Miss	b[0–7]
b[1]	0x20008	0b0010000000000000001000	0x10	0x0	Conflict Miss	c[0–7]
c[1]	0x30008	0b0011000000000000001000	0x18	0x0	Conflict Miss	d[0–7]
d[1]	0x40008	0b0100000000000000001000	0x20	0x0	Conflict Miss	e[0–7]
e[1]	0x50008	0b0101000000000000001000	0x28	0x0	Conflict Miss	a[0–7]

We have just visited 5 blocks after the last visit of a[0:7] — definitely not capacity miss

NVIDIA Tegra X1 (cont.)

- Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 8192; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

C = ABS
32KB = 4 * 64 * S
S = 128
offset = lg(64) = 6 bits
index = lg(128) = 7 bits
tag = the rest bits

tag index offset

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[7]	0x10038	0b0001000000000111000	0x8	0x0	**Conflict Miss**	
b[7]	0x20038	0b0010000000000111000	0x10	0x0	**Conflict Miss**	
c[7]	0x30038	0b0011000000000111000	0x18	0x0	**Conflict Miss**	
d[7]	0x40038	0b0100000000000111000	0x20	0x0	**Conflict Miss**	
e[7]	0x50038	0b0101000000000111000	0x28	0x0	**Conflict Miss**	a[0–7]
a[8]	0x10040	0b0001000000001000000	0x8	0x1	Compulsory Miss	
b[8]	0x20040	0b0010000000001000000	0x10	0x1	Compulsory Miss	
c[8]	0x30040	0b0011000000001000000	0x18	0x1	Compulsory Miss	
d[8]	0x40040	0b0100000000001000000	0x20	0x1	Compulsory Miss	
e[8]	0x50040	0b0101000000001000000	0x28	0x1	Compulsory Miss	a[8–15]
⋮	⋮	⋮	⋮	⋮	⋮	⋮

100% miss rate!

intel Core i7

- D-L1 Cache configuration of intel Core i7
  - Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 8192; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

How many of the cache misses are **compulsory** misses?

A. 12.5%
B. 66.67%
C. 68.75%
D. 87.5%
E. 100%

intel Core i7

- Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 8192; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

C = ABS
48KB = 12 * 64 * S
S = 64
offset = lg(64) = 6 bits
index = lg(64) = 6 bits
tag = the rest bits

tag index offset

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[0]	0x10000	0b00010000000000000000	0x10	0x0	Compulsory Miss	
b[0]	0x20000	0b00100000000000000000	0x20	0x0	Compulsory Miss	
c[0]	0x30000	0b00110000000000000000	0x30	0x0	Compulsory Miss	
d[0]	0x40000	0b01000000000000000000	0x40	0x0	Compulsory Miss	
e[0]	0x50000	0b01010000000000000000	0x50	0x0	Compulsory Miss	
a[1]	0x10008	0b00010000000000001000	0x10	0x0	Hit	
b[1]	0x20008	0b00100000000000001000	0x20	0x0	Hit	
c[1]	0x30008	0b00110000000000001000	0x30	0x0	Hit	
d[1]	0x40008	0b01000000000000001000	0x40	0x0	Hit	
e[1]	0x50008	0b01010000000000001000	0x50	0x0	Hit	
⋮	⋮	⋮	⋮	⋮	⋮	⋮

intel Core i7 (cont.)

- Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 8192; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

C = ABS
48KB = 12 * 64 * S
S = 64
offset = lg(64) = 6 bits
index = lg(64) = 6 bits
tag = the rest bits

tag index offset

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[7]	0x10038	0b0001000000000111000	0x10	0x0	Hit	
b[7]	0x20038	0b0010000000000111000	0x20	0x0	Hit	
c[7]	0x30038	0b0011000000000111000	0x30	0x0	Hit	
d[7]	0x40038	0b0100000000000111000	0x40	0x0	Hit	
e[7]	0x50038	0b0101000000000111000	0x50	0x0	Hit	
a[8]	0x10040	0b0001000000001000000	0x10	0x1	Compulsory Miss	
b[8]	0x20040	0b0010000000001000000	0x20	0x1	Compulsory Miss	
c[8]	0x30040	0b0011000000001000000	0x30	0x1	Compulsory Miss	
d[8]	0x40040	0b0100000000001000000	0x40	0x1	Compulsory Miss	
e[8]	0x50040	0b0101000000001000000	0x50	0x1	Compulsory Miss	
a[9]	0x10048	0b0001000000001001000	0x10	0x1	Hit	
b[9]	0x20048	0b0010000000001001000	0x20	0x1	Hit	
c[9]	0x30048	0b0011000000001001000	0x30	0x1	Hit	
d[9]	0x40048	0b0100000000001001000	0x40	0x1	Hit	

intel Core i7

- D-L1 Cache configuration of intel Core i7
  - Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 8192; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

How many of the cache misses are **compulsory** misses?

A. 12.5%
B. 66.67%
C. 68.75%
D. 87.5%
E. 100%

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

https://www.pollev.com/hungweitseng close in 1:30

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

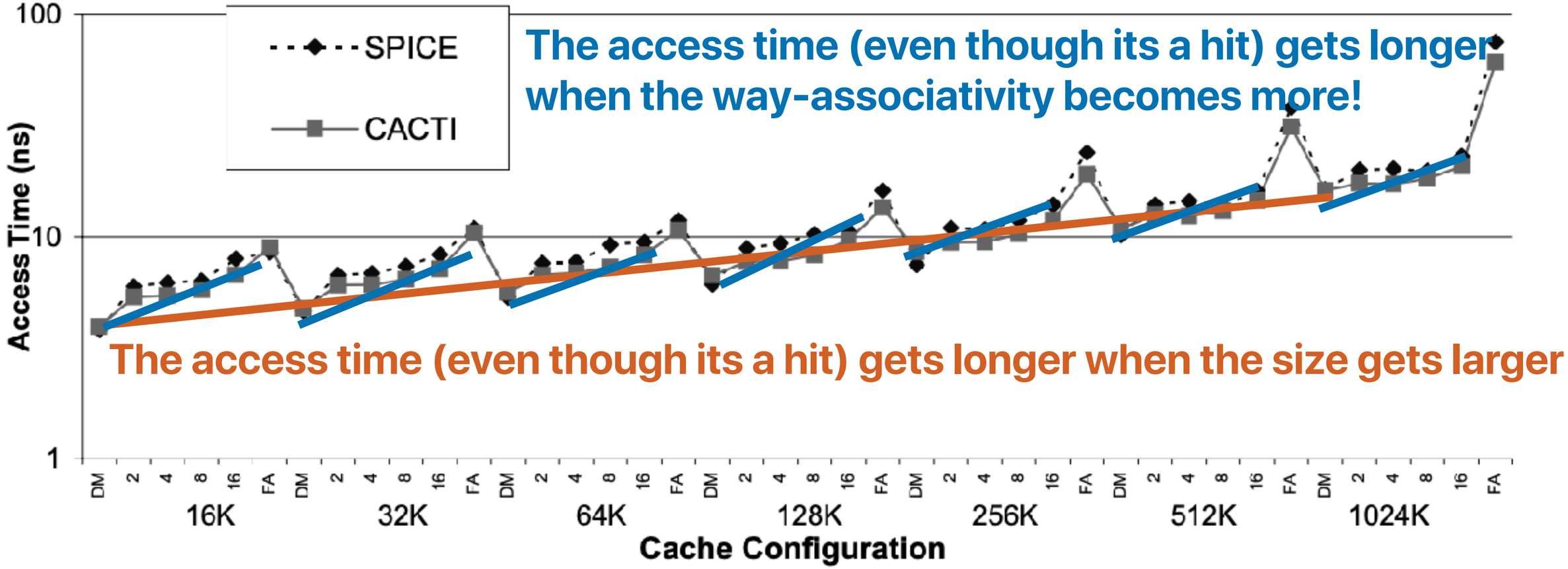
  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

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

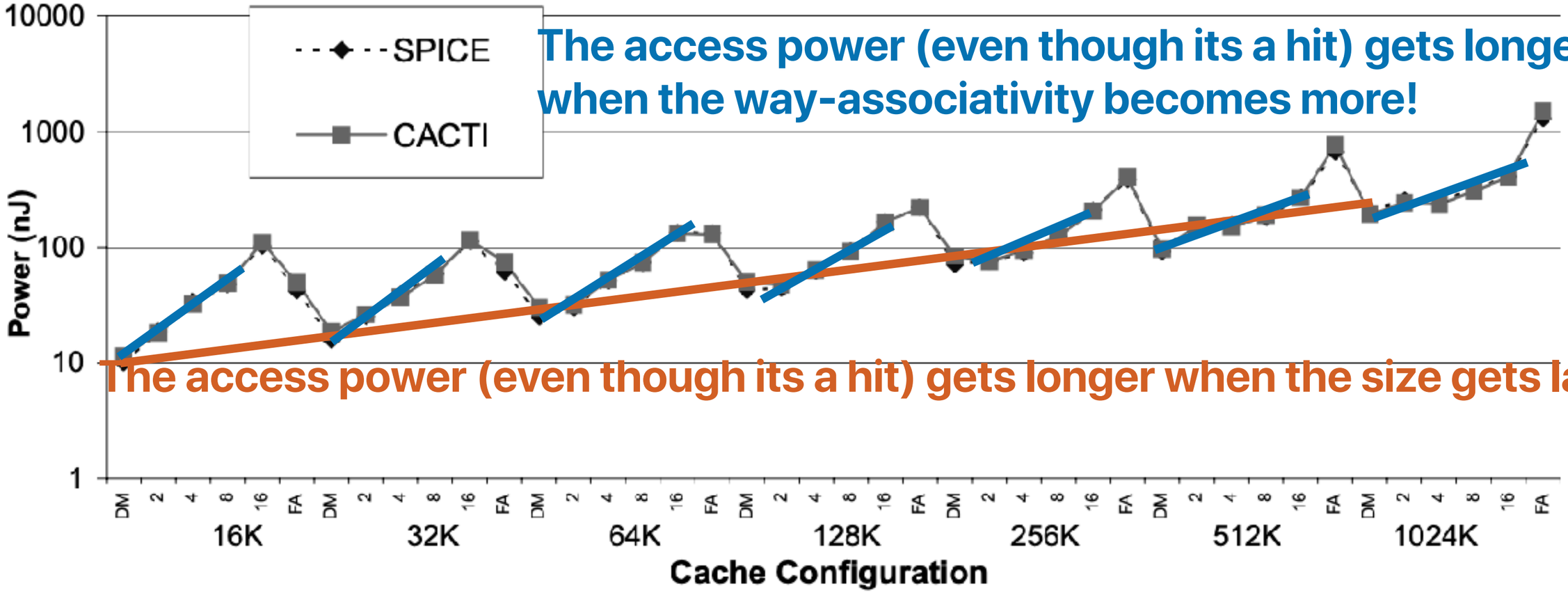
  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

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?

Recap: NVIDIA Tegra X1

- D-L1 Cache configuration of NVIDIA Tegra X1
  - Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

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

A. 12.5%
B. 56.25%
C. 66.67%
D. 68.75%
E. 100%

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

Norman P. Jouppi

https://www.pollev.com/hungweitseng close in 1:30

Which of the following schemes can help NVIDIA Tegra?

- How many of the following schemes mentioned in “improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers” would help NVIDIA’s Tegra for the code in the previous slide?
  - ① Missing cache
  - ② Victim cache
  - ③ Prefetch
  - ④ Stream buffer

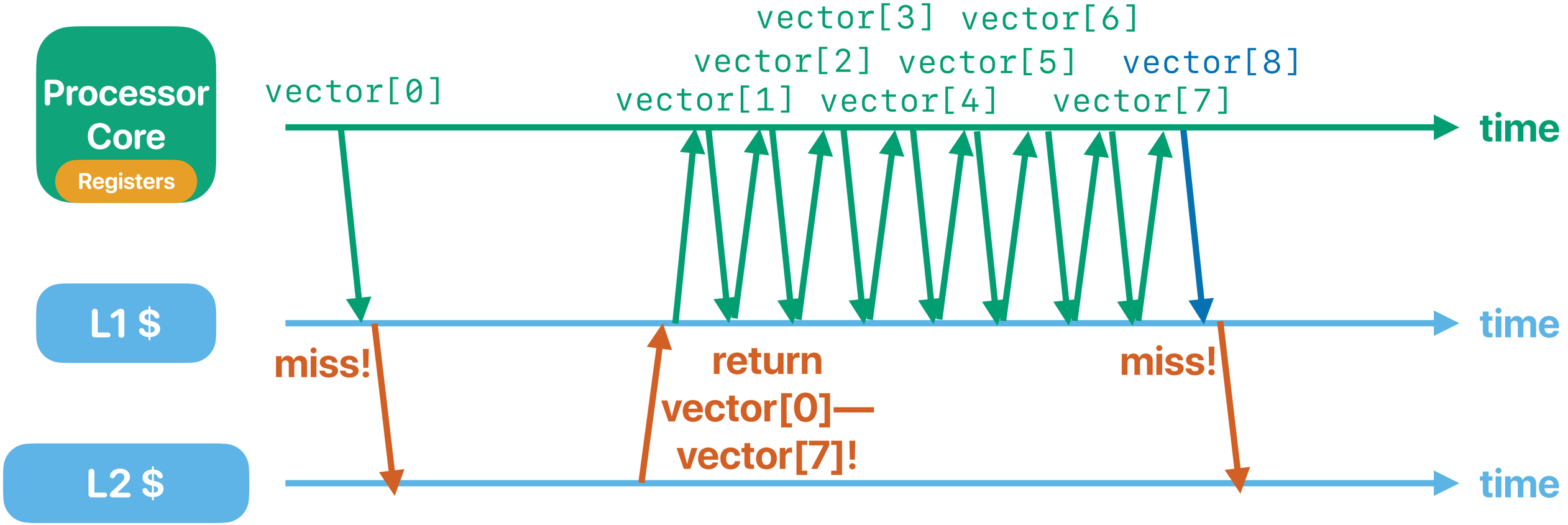
A. 0
B. 1
C. 2
D. 3
E. 4

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 8192; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```

Prefetching

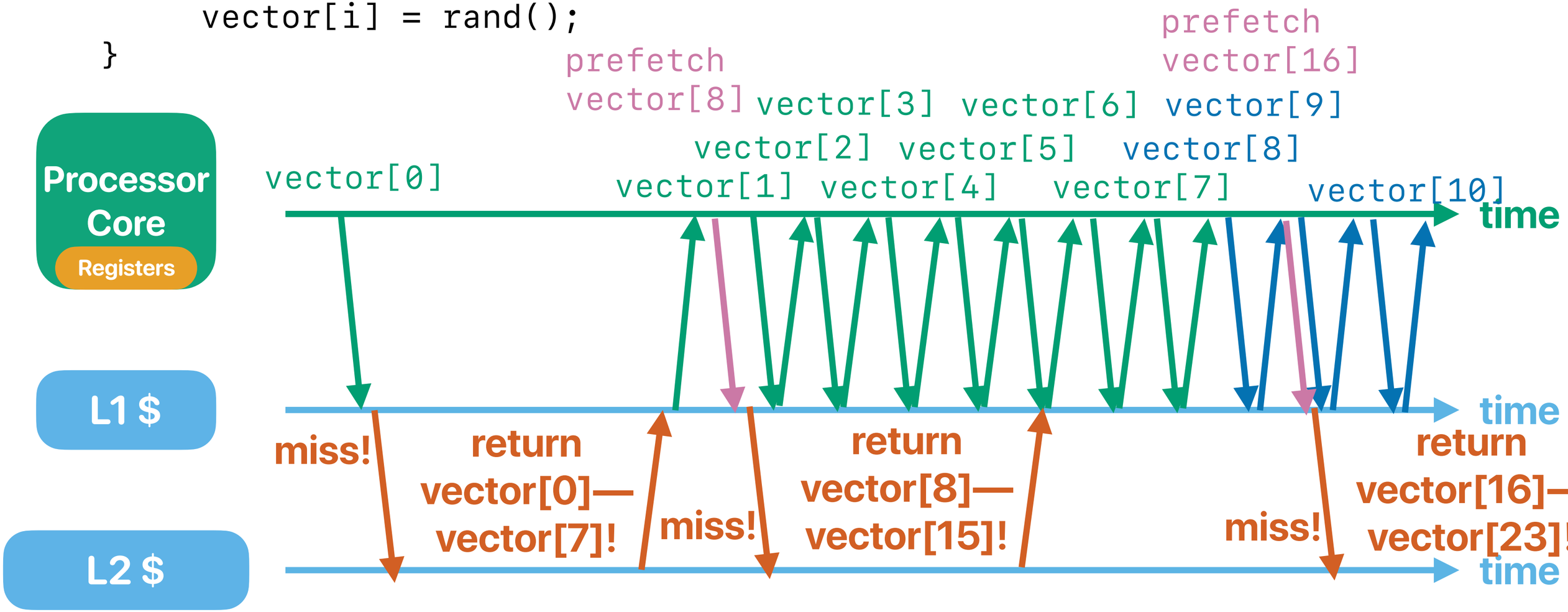
Spatial locality revisited

```
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

Where can prefetch work effectively?

- How many of the following code snippet can “prefetching” effectively help improving performance?

(1)
```
while(node){
     node = node->next;
}
```

(2)
```
while(++i<100000)
     a[i]=rand();
```

(3)
```
while (root != NULL){
        if (key > root->data)
            root = root->right;

        else if (key < root->data)
            root = root->left;
        else
            return true;
}
```

(4)
```
    for (i = 0; i < 65536; i++) {
      mix_i = ((i * 167) + 13) & 65536;
      results[mix_i]++;
    }
```

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

Where can prefetch work effectively?

- How many of the following code snippet can “prefetching” effectively help improving performance?

(1)
```
while(node){
    node = node->next;
}
```
— where the next pointing to is hard to predict

(2)
```
while(++i<100000)
    a[i]=rand();
```

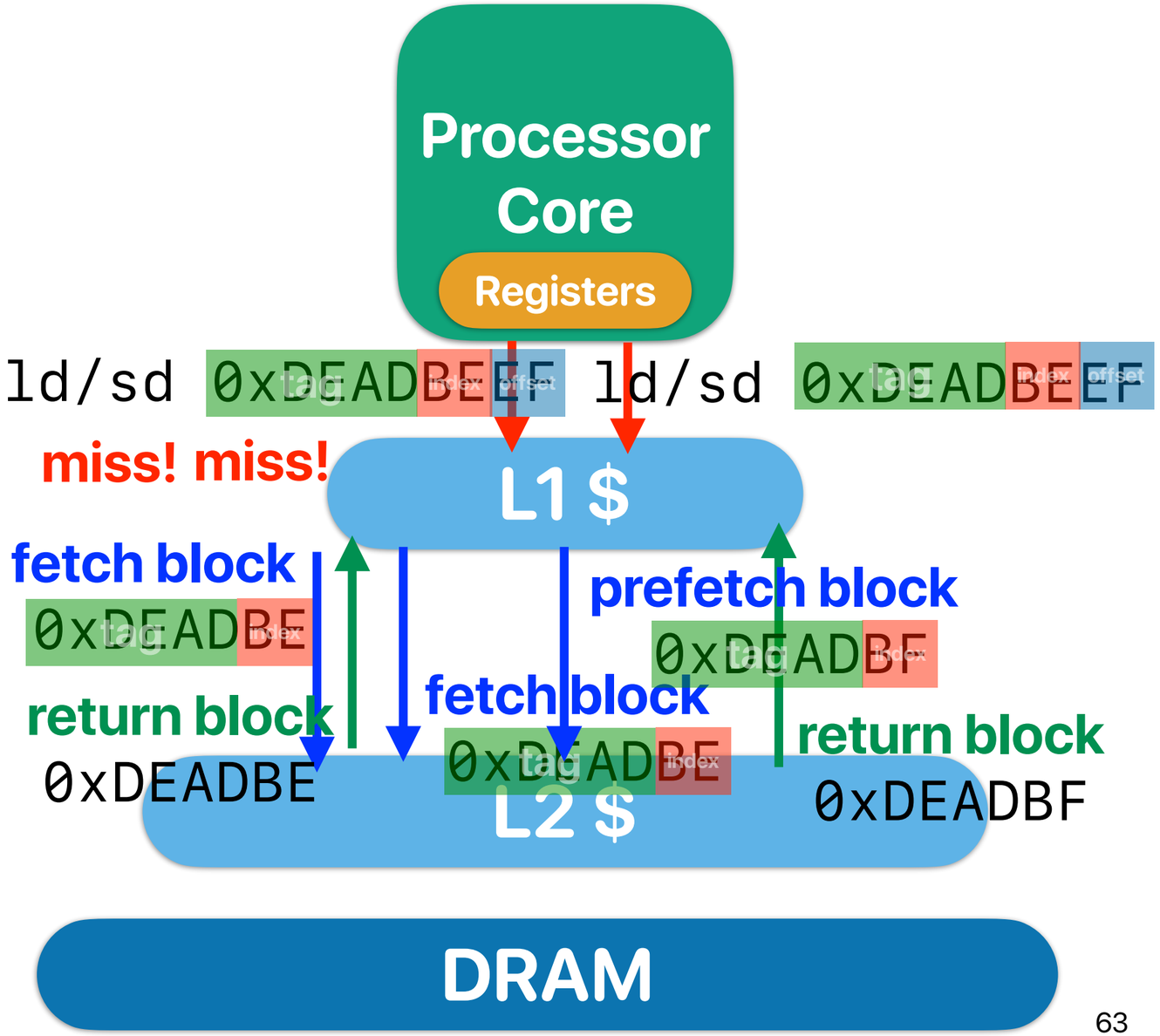
(3)
```
while (root != NULL){
        if (key > root->data)
            root = root->right;

        else if (key < root->data)
            root = root->left;
        else
            return true;
}
```
— where the next node is also hard to predict

(4)
```
    for (i = 0; i < 65536; i++) {
      mix_i = ((i * 167) + 13) & 65536;
      results[mix_i]++;
    }
```
— the stride to the next element is hard to predict…

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

What's after prefetching?



NVIDIA Tegra X1 with prefetch

- Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

C = ABS
32KB = 4 * 64 * S
S = 128
offset = lg(64) = 6 bits
index = lg(128) = 7 bits
tag = the rest bits

tag index offset

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?	Prefetch
a[0]	0x10000	0b0001000000000000000000	0x8	0x0	Miss		a[8–15]
b[0]	0x20000	0b0010000000000000000000	0x10	0x0	Miss		b[8–15]
c[0]	0x30000	0b0011000000000000000000	0x18	0x0	Miss		c[8–15]
d[0]	0x40000	0b0100000000000000000000	0x20	0x0	Miss		d[8–15]
e[0]	0x50000	0b0101000000000000000000	0x28	0x0	Miss	a[0–7]	e[8–15]
a[1]	0x10008	0b0001000000000000001000	0x8	0x0	Miss	b[0–7]	
b[1]	0x20008	0b0010000000000000001000	0x10	0x0	Miss	c[0–7]	
c[1]	0x30008	0b0011000000000000001000	0x18	0x0	Miss	d[0–7]	
d[1]	0x40008	0b0100000000000000001000	0x20	0x0	Miss	e[0–7]	
e[1]	0x50008	0b0101000000000000001000	0x28	0x0	Miss	a[0–7]	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

e[8-15] will kick out a[8-15]

100% miss rate!

Announcement

- Reading quiz #4 due next **Tuesday** before the lecture
- Upcoming deadlines
  - Assignment #2 due **in next Thursday — 10/24/2024**
    - Already online — please find through the course website
    - Don’t submit the wrong one
  - Programming Assignment #2 due **in 4 weeks — 11/7/2024**
    - Already online — please find through the course website
    - Don’t submit the wrong one
  - Assignment #3 due **in three weeks — 10/31/2024**
    - Will be online next Thursday— please find through the course website
    - Don’t submit the wrong one

Computer
Science &
Engineering

203

つづく