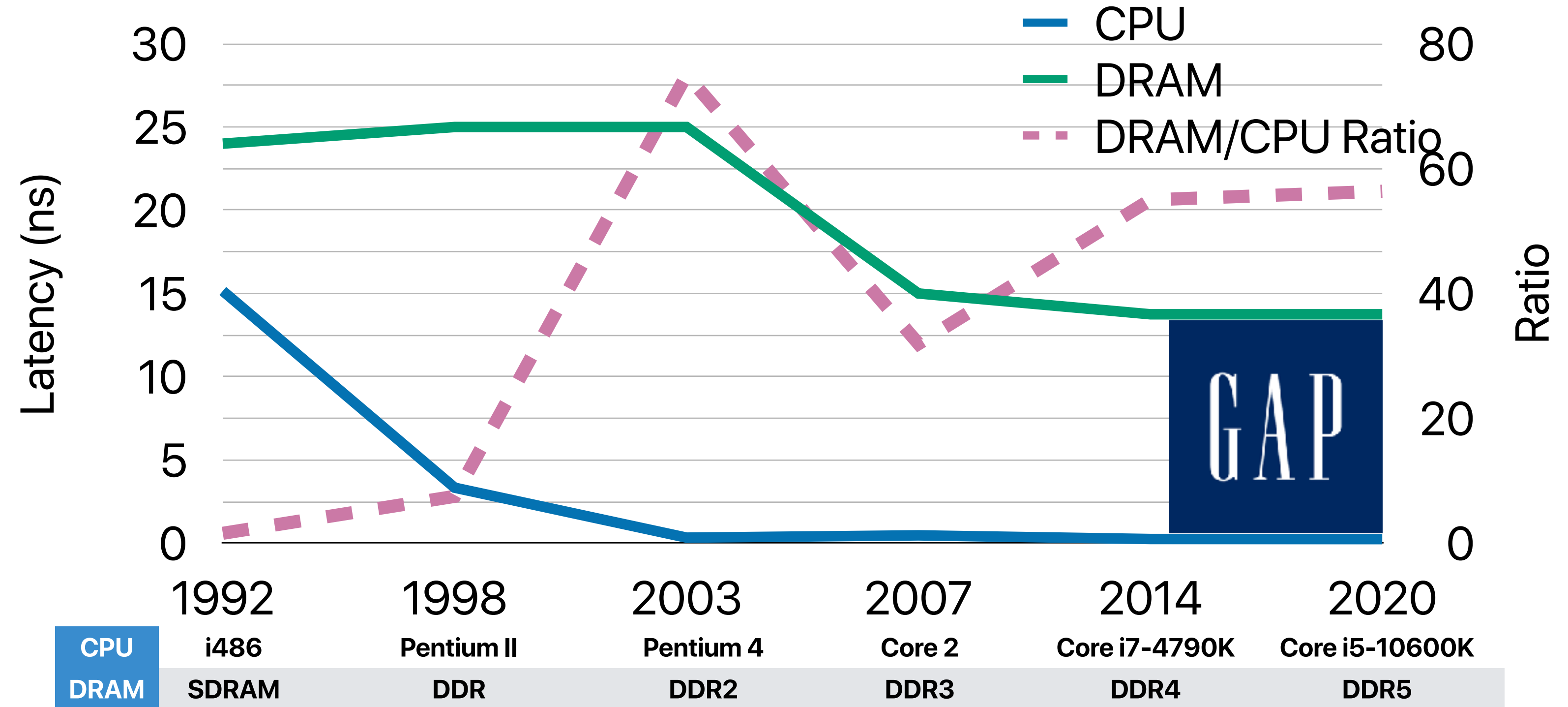


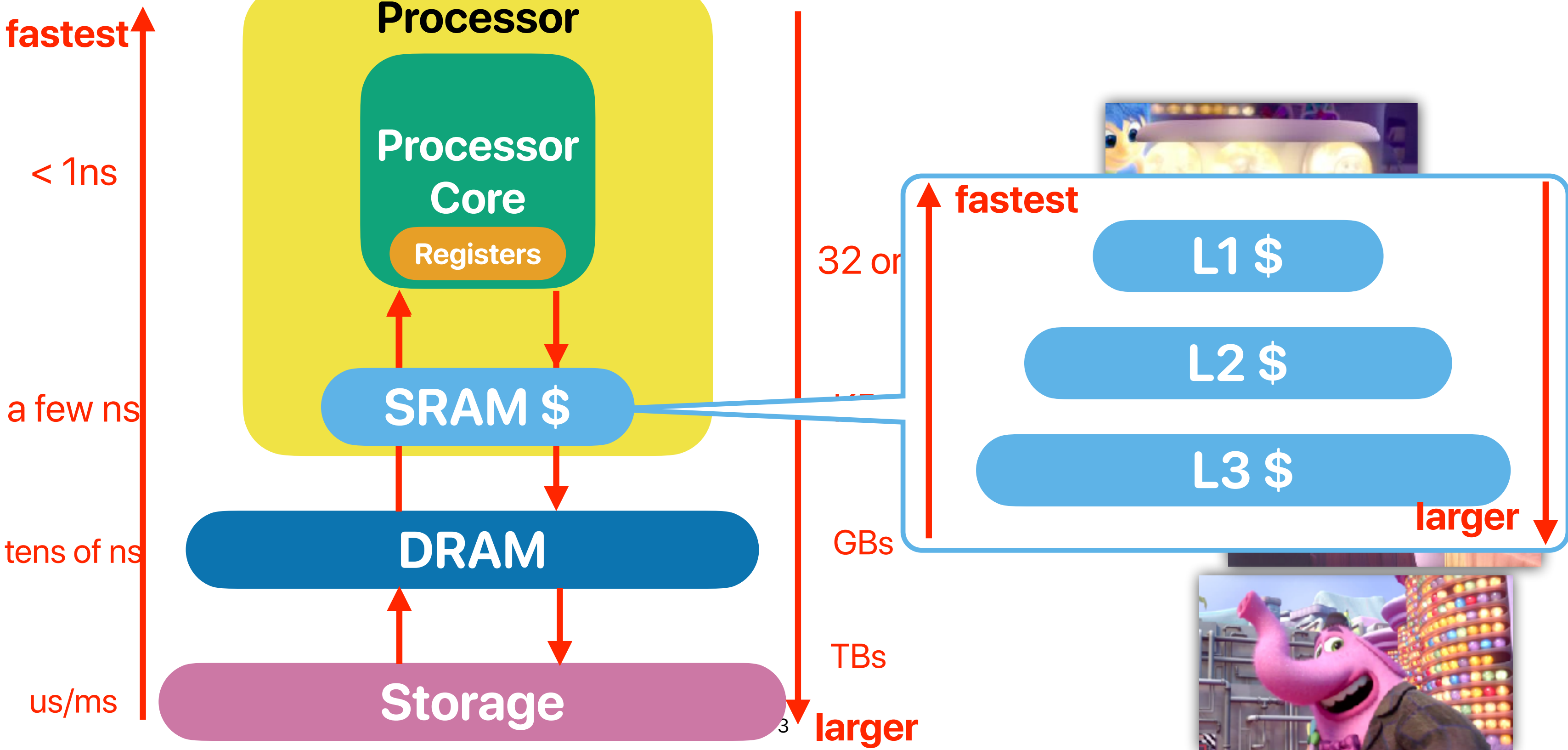
Memory hierarchy (3): the causes of cache misses and how to address them — the hardware version

Hung-Wei Tseng

The "latency" gap between CPU and DRAM

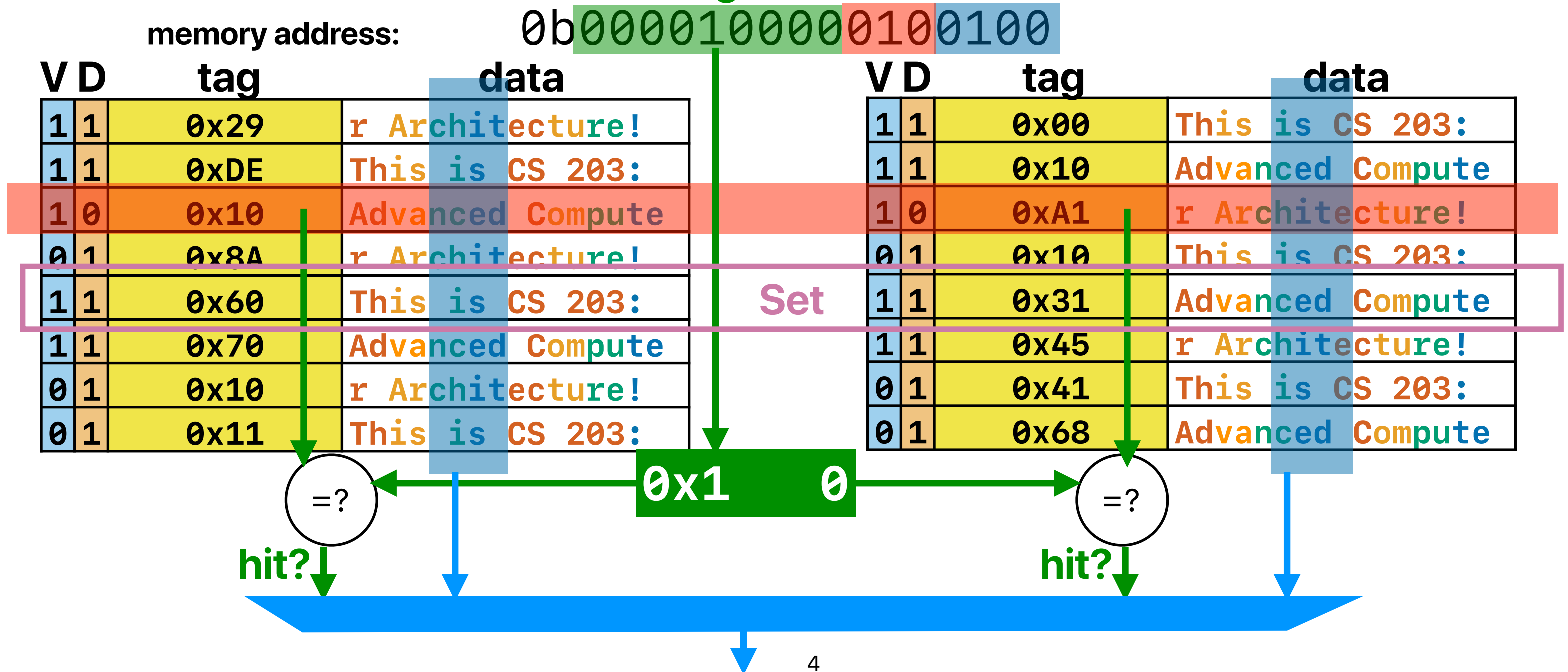


Recap: Memory Hierarchy



Way-associative cache

memory address: 0x0 8 2 4
tag set index block offset



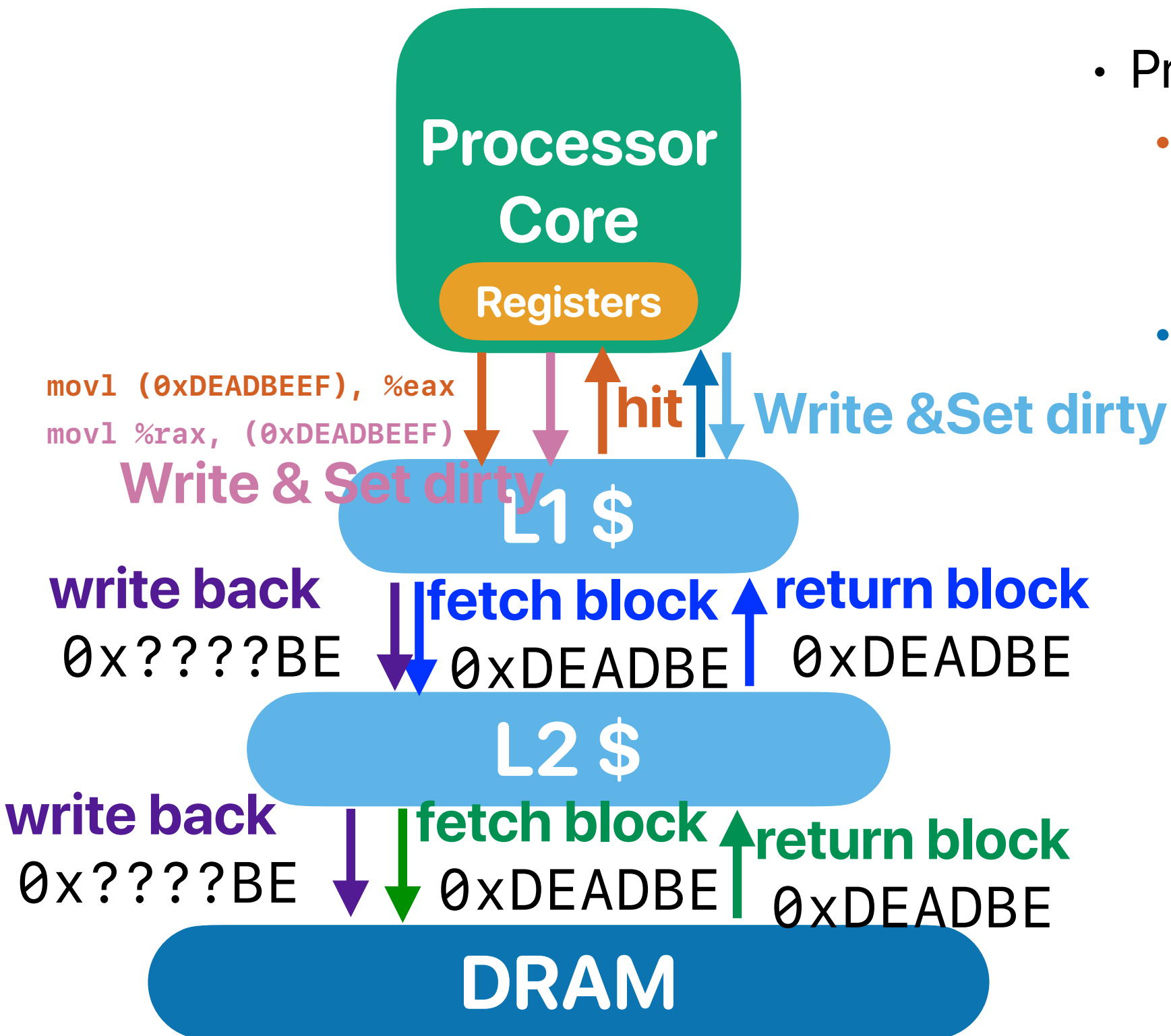
Recap: $C = ABS$

- **C**: **C**apacity in data arrays
- **A**: **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
- number of bits in **b**lock offset — $\lg(\mathbf{B})$
- number of bits in **s**et index: $\lg(\mathbf{S})$
- tag bits: $\text{address_length} - \lg(\mathbf{S}) - \lg(\mathbf{B})$
 - address_length is N bits for N-bit machines (e.g., 64-bit for 64-bit machines)
- $(\text{address} / \text{block_size}) \% S = \text{set index}$

memory address:

0b tag set block
index offset
00001000000100100

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**

NVIDIA Tegra Orin

100% miss rate!

- 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], d[16384], e[16384];
/* 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
64KB = 4 * 64 * S
S = 256
offset = lg(64) = 6 bits
index = lg(256) = 8 bits
tag = the rest bits

	Address (Hex)	Address in binary			Tag	Index	Hit? Miss?	Replace?
a[0]	0x10000	0b000100	00000000	0000000	0x4	0x0	Miss	
b[0]	0x20000	0b001000	00000000	0000000	0x8	0x0	Miss	
c[0]	0x30000	0b001100	00000000	0000000	0xC	0x0	Miss	
d[0]	0x40000	0b010000	00000000	0000000	0x10	0x0	Miss	
e[0]	0x50000	0b010100	00000000	0000000	0x14	0x0	Miss	a[0-7]
a[1]	0x10008	0b000100	00000000	001000	0x4	0x0	Miss	b[0-7]
b[1]	0x20008	0b001000	00000000	001000	0x8	0x0	Miss	c[0-7]
c[1]	0x30008	0b001100	00000000	001000	0xC	0x0	Miss	d[0-7]
d[1]	0x40008	0b010000	00000000	001000	0x10	0x0	Miss	e[0-7]
e[1]	0x50008	0b010100	00000000	001000	0x14	0x0	Miss	a[0-7]
⋮	⋮	⋮			⋮	⋮	⋮	⋮

NVIDIA Tegra Orin

- 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], d[16384], e[16384];  
/* 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%

During examines/interviews, why do you (occasionally) don't know the answer?



intel Core i7 14700K

- D-L1 Cache configuration of intel Core i7 14700K's P-Core
 - Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* 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%



intel Core i7 14700K

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

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* 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
48KB = 12 * 64 * S
S = 64
offset = lg(64) = 6 bits
index = lg(12) = 3 bits
tag = the rest bits

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[0]	0x10000	0b0001000000000000000000	0x10	0x0	Miss	
b[0]	0x20000	0b0010000000000000000000	0x20	0x0	Miss	
c[0]	0x30000	0b0011000000000000000000	0x30	0x0	Miss	
d[0]	0x40000	0b0100000000000000000000	0x40	0x0	Miss	
e[0]	0x50000	0b0101000000000000000000	0x50	0x0	Miss	
a[1]	0x10008	0b0001000000000000001000	0x10	0x0	Hit	
b[1]	0x20008	0b0010000000000000001000	0x20	0x0	Hit	
c[1]	0x30008	0b0011000000000000001000	0x30	0x0	Hit	
d[1]	0x40008	0b0100000000000000001000	0x40	0x0	Hit	
e[1]	0x50008	0b0101000000000000001000	0x50	0x0	Hit	
⋮	⋮	⋮	⋮	⋮	⋮	⋮

intel Core i7 14700K (cont.)

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

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* 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
48KB = 12 * 64 * S
S = 64
offset = lg(64) = 6 bits
index = lg(64) = 6 bits
tag = the rest bits

	Address (Hex)	Address in binary			Tag	Index	Hit? Miss?	Replace?
a[7]	0x10038	0b00010000	000000	111000	0x10	0x0	Hit	
b[7]	0x20038	0b00100000	000000	111000	0x20	0x0	Hit	
c[7]	0x30038	0b00110000	000000	111000	0x30	0x0	Hit	
d[7]	0x40038	0b01000000	000000	111000	0x40	0x0	Hit	
e[7]	0x50038	0b01010000	000000	111000	0x50	0x0	Hit	
a[8]	0x10040	0b00010000	000001	000000	0x10	0x1	Miss	
b[8]	0x20040	0b00100000	000001	000000	0x20	0x1	Miss	
c[8]	0x30040	0b00110000	000001	000000	0x30	0x1	Miss	
d[8]	0x40040	0b01000000	000001	000000	0x40	0x1	Miss	
e[8]	0x50040	0b01010000	000001	000000	0x50	0x1	Miss	
a[9]	0x10048	0b00010000	000001	001000	0x10	0x1	Hit	
b[9]	0x20048	0b00100000	000001	001000	0x20	0x1	Hit	
c[9]	0x30048	0b00110000	000001	001000	0x30	0x1	Hit	
d[9]	0x40048	0b01000000	000001	001000	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!

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
 - The working set size is the size of data blocks visited between the last observation point and the current point — **not** the volume of your data structure
- Conflict miss
 - Required data replaced by block(s) mapping to the same set
 - Similar collision in hash
 - The working set size is smaller than the cache capacity



NVIDIA Tegra Orin

- 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], d[16384], e[16384];
/* 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 major type of cache misses when running the above code snippet?

- A. Compulsory misses
- B. Conflict misses
- C. Capacity misses
- D. Capacity and conflict misses
- E. They are all equally critical



NVIDIA Tegra Orin

100% miss rate!

- 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], d[16384], e[16384];
/* 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
64KB = 4 * 64 * S
S = 256
offset = lg(64) = 6 bits
index = lg(256) = 8 bits
tag = the rest bits

	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]
⋮	⋮	⋮	⋮	⋮	⋮	⋮

NVIDIA Tegra Orin (cont.)

- 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], d[16384], e[16384];
/* 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
64KB = 4 * 64 * S
S = 256
offset = lg(64) = 6 bits
index = lg(256) = 8 bits
tag = the rest bits

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[7]	0x10038	0b00010000000000111000	0x8	0x0	Conflict Miss	
b[7]	0x20038	0b00100000000000111000	0x10	0x0	Conflict Miss	
c[7]	0x30038	0b00110000000000111000	0x18	0x0	Conflict Miss	
d[7]	0x40038	0b01000000000000111000	0x20	0x0	Conflict Miss	
e[7]	0x50038	0b01010000000000111000	0x28	0x0	Conflict Miss	a[0-7]
a[8]	0x10040	0b00010000000001000000	0x8	0x1	Compulsory Miss	
b[8]	0x20040	0b00100000000001000000	0x10	0x1	Compulsory Miss	
c[8]	0x30040	0b00110000000001000000	0x18	0x1	Compulsory Miss	
d[8]	0x40040	0b01000000000001000000	0x20	0x1	Compulsory Miss	
e[8]	0x50040	0b01010000000001000000	0x28	0x1	Compulsory Miss	a[8-15]

100% miss rate!

NVIDIA Tegra Orin

- 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], d[16384], e[16384];
/* 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 major type of cache misses when running the above code snippet?

- A. Compulsory misses
- B. Conflict misses
- C. Capacity misses
- D. Capacity and conflict misses
- E. They are all equally critical



intel Core i7 14700K

- D-L1 Cache configuration of intel Core i7 14700K's P-core
 - Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* 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 major type of cache misses when running the above code snippet?

- A. Compulsory misses
- B. Conflict misses
- C. Capacity misses
- D. Capacity and conflict misses
- E. They are all equally critical



intel Core i7 14700K

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

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* 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
 48KB = 12 * 64 * S
 S = 64
 offset = lg(64) = 6 bits
 index = lg(12) = 3 bits
 tag = the rest bits

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[0]	0x10000	0b0001000000000000000000	0x10	0x0	Compulsory Miss	
b[0]	0x20000	0b0010000000000000000000	0x20	0x0	Compulsory Miss	
c[0]	0x30000	0b0011000000000000000000	0x30	0x0	Compulsory Miss	
d[0]	0x40000	0b0100000000000000000000	0x40	0x0	Compulsory Miss	
e[0]	0x50000	0b0101000000000000000000	0x50	0x0	Compulsory Miss	
a[1]	0x10008	0b0001000000000000001000	0x10	0x0	Hit	
b[1]	0x20008	0b0010000000000000001000	0x20	0x0	Hit	
c[1]	0x30008	0b0011000000000000001000	0x30	0x0	Hit	
d[1]	0x40008	0b0100000000000000001000	0x40	0x0	Hit	
e[1]	0x50008	0b0101000000000000001000	0x50	0x0	Hit	
⋮	⋮	⋮	⋮	⋮	⋮	⋮

intel Core i7 14700K (cont.)

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

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* 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
48KB = 12 * 64 * S
S = 64
offset = lg(64) = 6 bits
index = lg(64) = 6 bits
tag = the rest bits

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[7]	0x10038	0b00010000000000111000	0x10	0x0	Hit	
b[7]	0x20038	0b00100000000000111000	0x20	0x0	Hit	
c[7]	0x30038	0b00110000000000111000	0x30	0x0	Hit	
d[7]	0x40038	0b01000000000000111000	0x40	0x0	Hit	
e[7]	0x50038	0b01010000000000111000	0x50	0x0	Hit	
a[8]	0x10040	0b00010000000001000000	0x10	0x1	Compulsory Miss	
b[8]	0x20040	0b00100000000001000000	0x20	0x1	Compulsory Miss	
c[8]	0x30040	0b00110000000001000000	0x30	0x1	Compulsory Miss	
d[8]	0x40040	0b01000000000001000000	0x40	0x1	Compulsory Miss	
e[8]	0x50040	0b01010000000001000000	0x50	0x1	Compulsory Miss	
a[9]	0x10048	0b00010000000001001000	0x10	0x1	Hit	
b[9]	0x20048	0b00100000000001001000	0x20	0x1	Hit	
c[9]	0x30048	0b00110000000001001000	0x30	0x1	Hit	
d[9]	0x40048	0b01000000000001001000	0x40	0x1	Hit	

intel Core i7 14700K

- D-L1 Cache configuration of intel Core i7 14700K's P-core
 - Size 48KB, 12-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
double a[16384], b[16384], c[16384], d[16384], e[16384];
/* 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 major type of cache misses when running the above code snippet?

- A. Compulsory misses
- B. Conflict misses
- C. Capacity misses
- D. Capacity and conflict misses
- E. They are all equally critical

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
 - Capacity misses — the miss due to the working set size surpasses the **capacity**
 - Conflict misses — the miss due to insufficient blocks of the target set (**associativity**)

A, B, C, S and cache misses



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

A screenshot of a poll interface. It shows five horizontal input boxes, each preceded by a letter (A, B, C, D, E) in a small font. The boxes are empty, suggesting a multiple-choice or short-answer poll.

3Cs and A, B, C

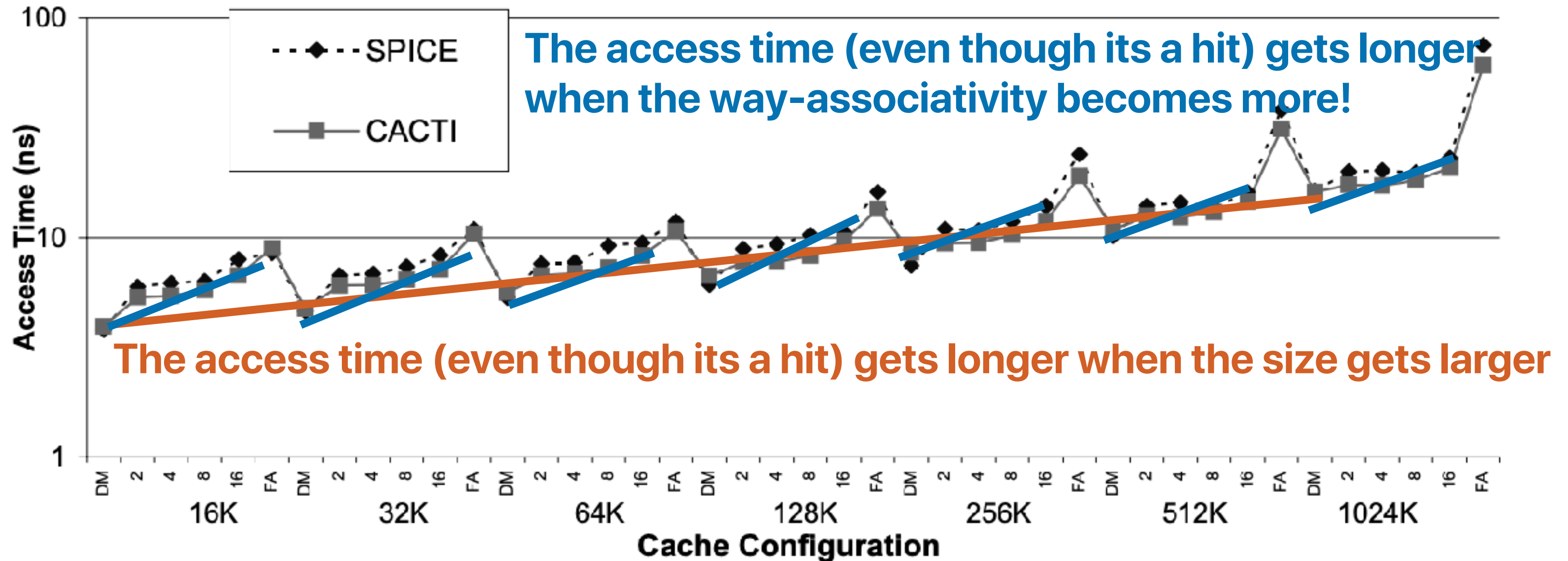
- Regarding 3Cs: compulsory, conflict and capacity misses and
A, B, C: associativity, block size, capacity

How many of the following are correct?

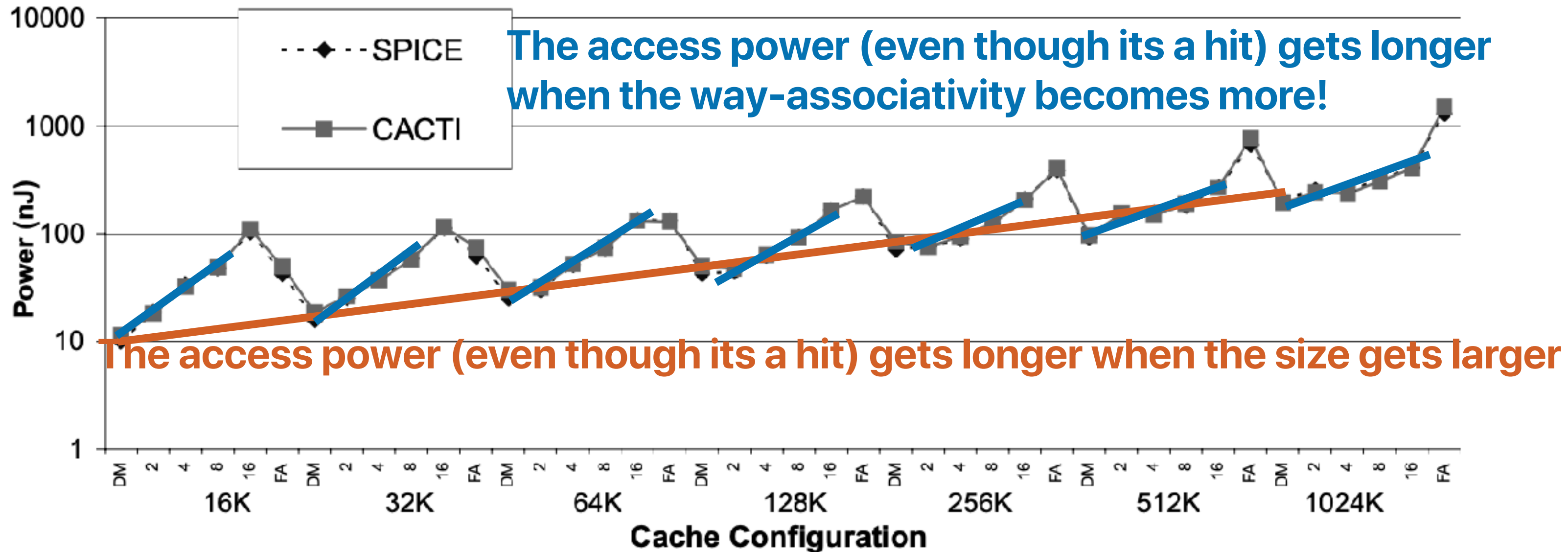
- ① Without changing B & C, increasing A can reduce conflict misses but make each cache hit slower
- ② Without changing A & C, increasing B can reduce compulsory misses but potentially lead to more conflict misses
- ③ Without changing A & C, increasing B will make each cache miss slower
- ④ Without changing 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



Cache configurations and accessing power



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





Which of the following schemes can help NVIDIA Orin?

- 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

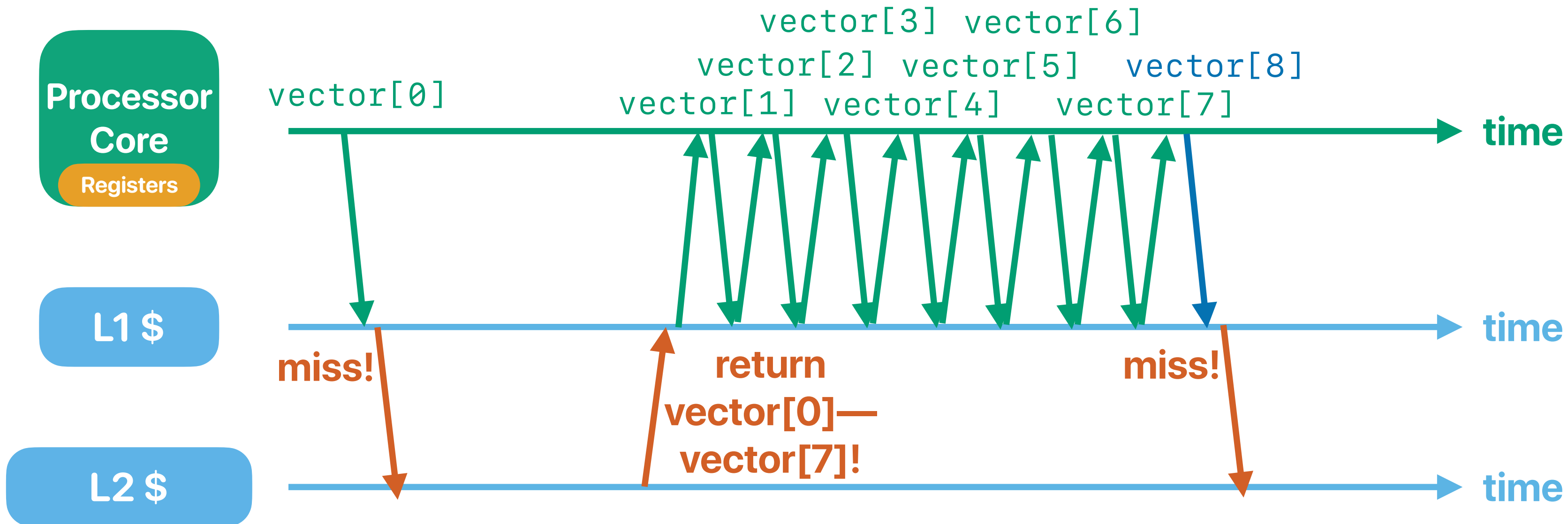
```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 8192; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```



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();  
}
```

prefetch
vector[8]

prefetch
vector[16]

vector[3] vector[6] vector[9]

vector[2] vector[5] vector[8]

vector[1] vector[4] vector[7] vector[10]

time

Processor
Core

Registers

L1 \$

time

miss!

return
vector[0]—
vector[7]!

miss!

return
vector[8]—
vector[15]!

miss!

return
vector[16]—
vector[23]!

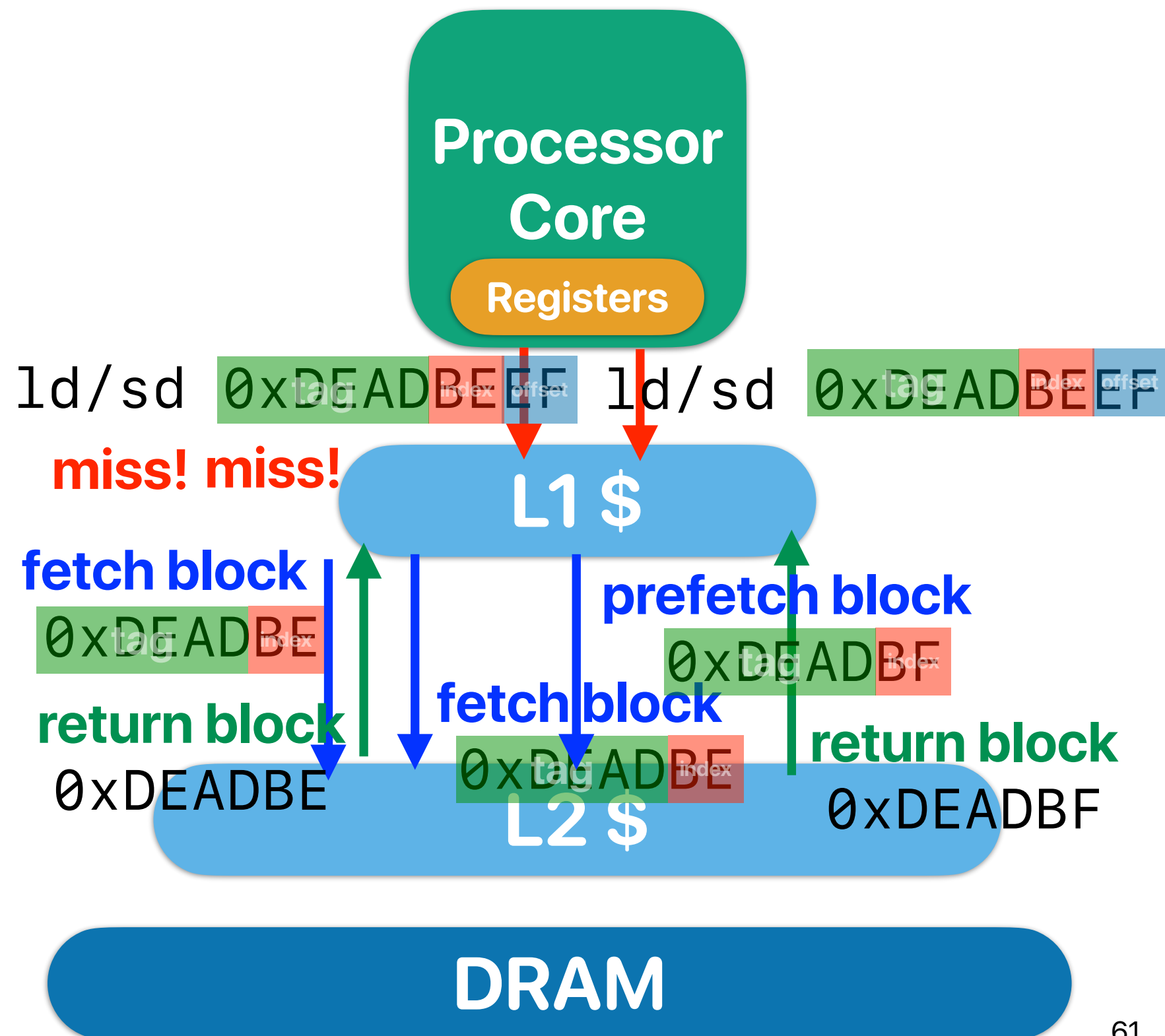
time

L2 \$

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 $\text{miss_data_address} + \text{offset}$ for a miss

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.

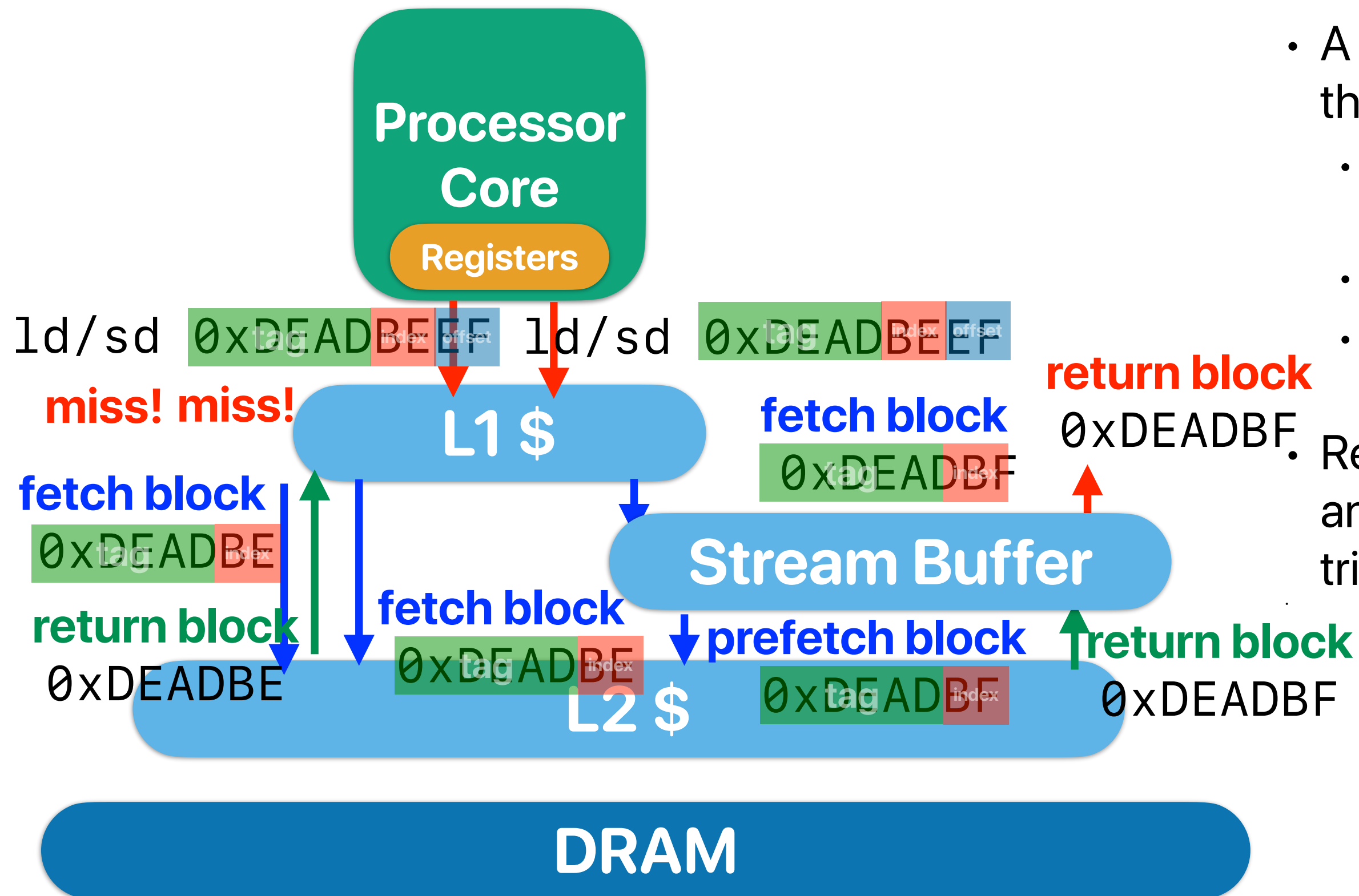
```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

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

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

100% miss rate!

Stream buffer

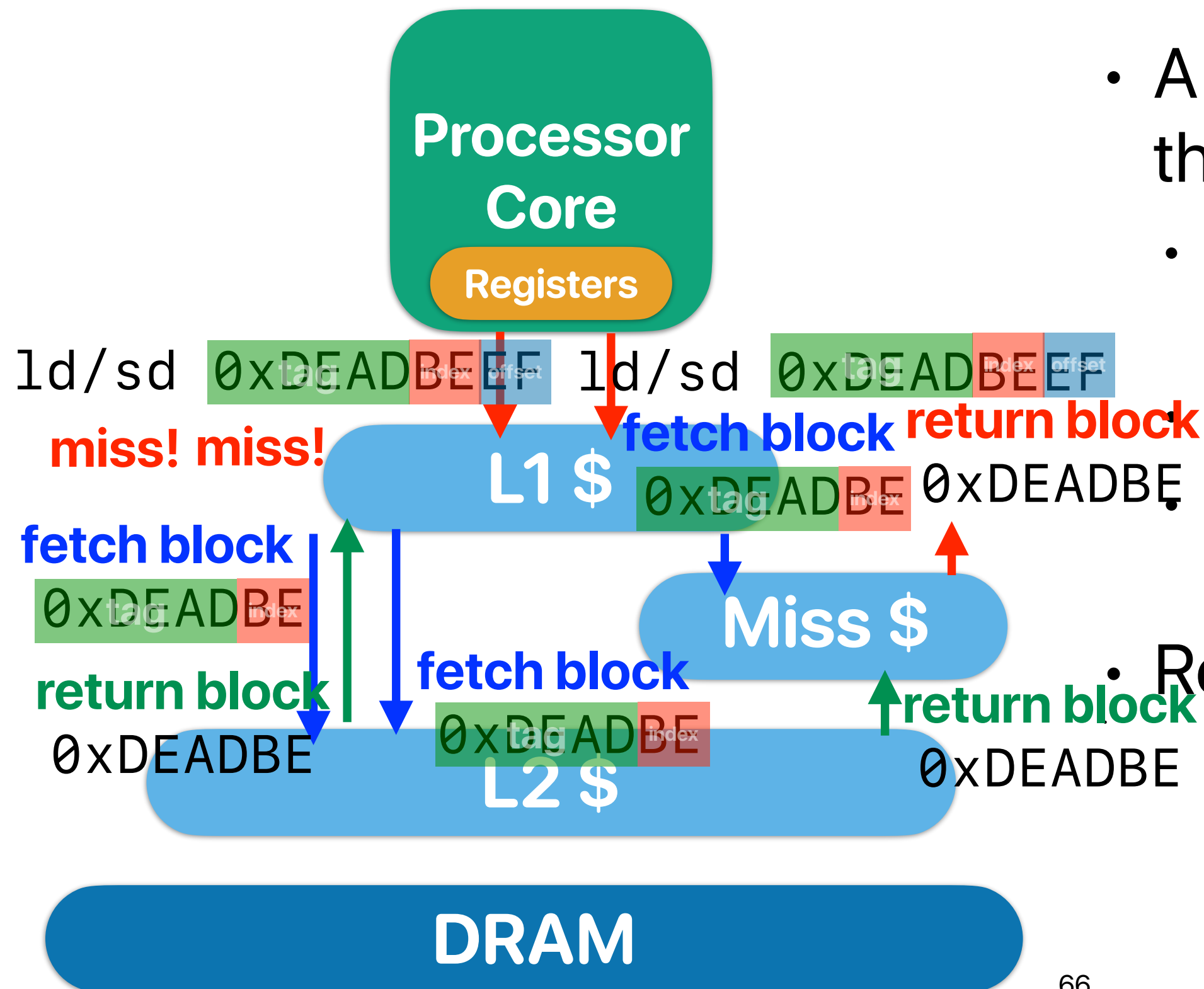


- A small cache that captures the prefetched blocks
 - Can be built as fully associative since it's small
 - Consult when there is a miss
 - Retrieve the block if found in the stream buffer
- Reduce compulsory misses and avoid conflict misses triggered by prefetching

Software Prefetching — through prefetching instructions

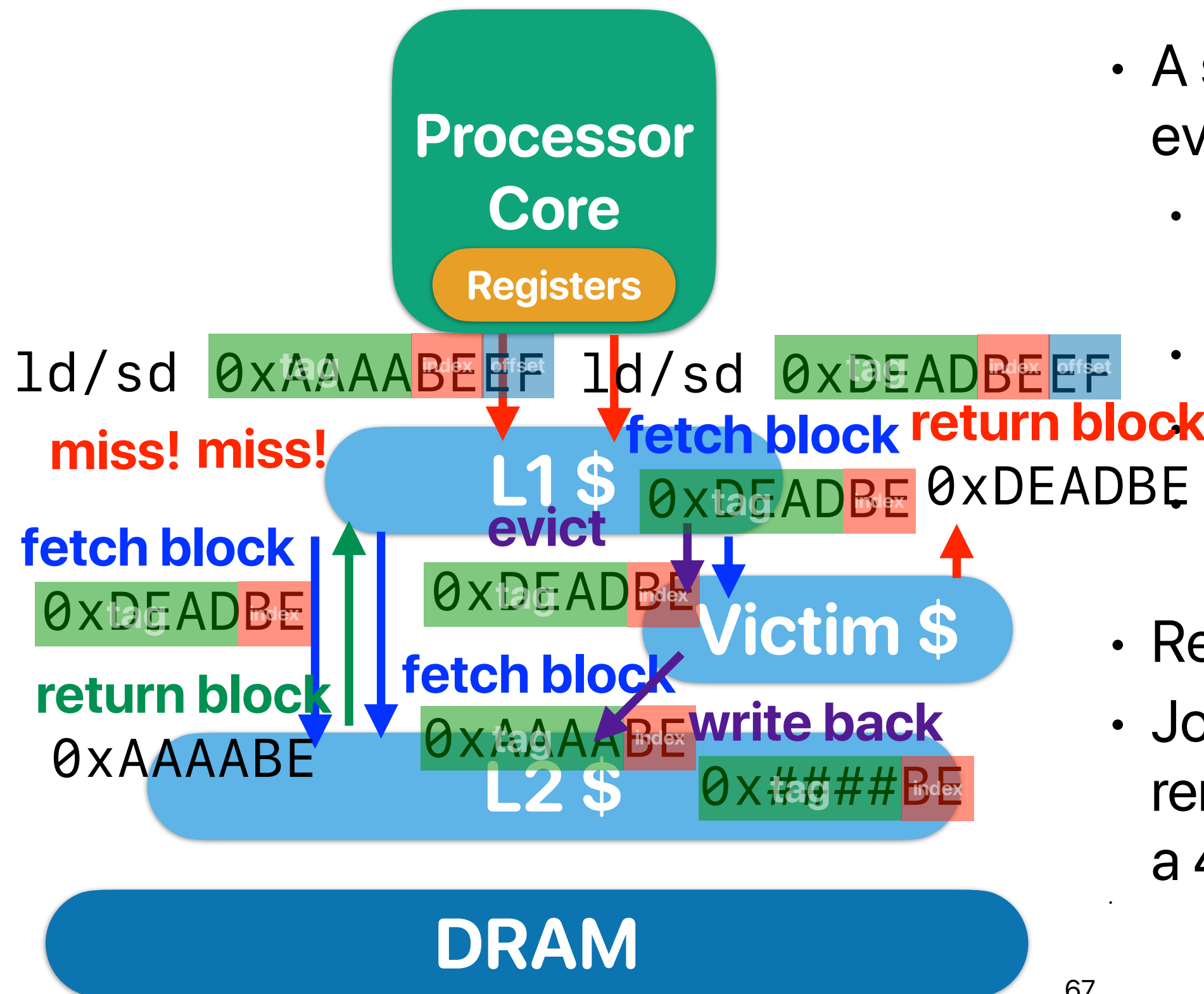
- x86 provide prefetch instructions
- As a programmer, you may insert `_mm_prefetch` in x86 programs to perform software prefetch for your code
- gcc also has a flag `"-fprefetch-loop-arrays"` to automatically insert software prefetch instructions

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

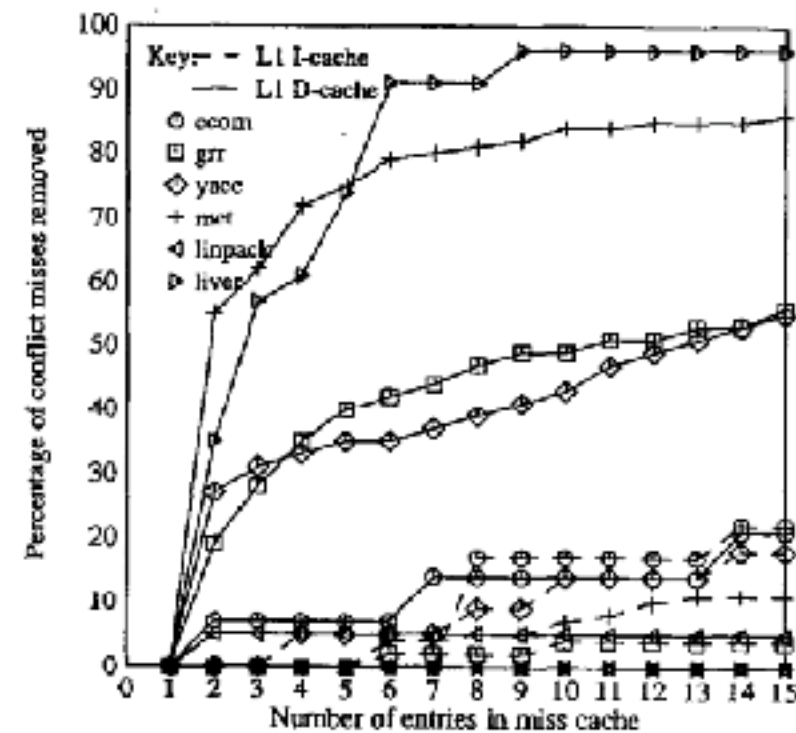


Figure 3-3: Conflict misses removed by miss caching

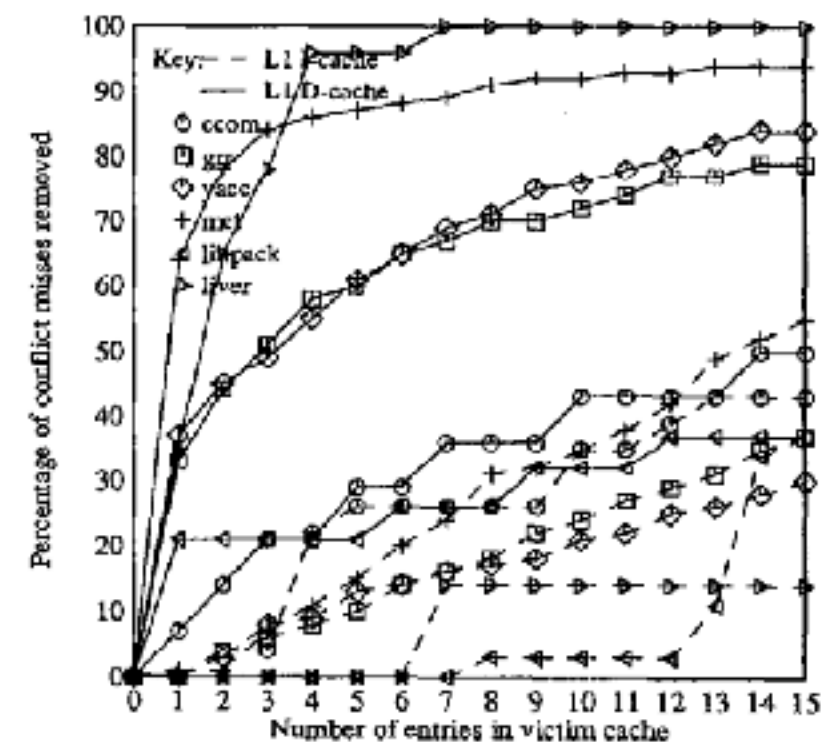


Figure 3-5: Conflict misses removed by victim caching

Which of the following schemes can help 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 — help improving conflict misses
 - ② ☒ Victim cache — help improving conflict misses
 - ③ ☐ Prefetch — improving compulsory misses , but can potentially hurt, if we did not do it right
 - ④ ☒ Stream buffer — only help improving compulsory misses
- A. 0
B. 1
C. 2
D. 3
E. 4

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

Announcement

- Reading quiz #4 due next **Tuesday** before the lecture
- Assignment #2 released and due **next Thursday**
 - We do not support people on the last minute. Please carefully plan your time and respect the instructor/TA
- Programming Assignment 2 is performance-based, please start early
- Midterm/Final is exactly the same environment as the testing center provides
 - No external calculator
 - No external pen & papers
 - Calculator is only available under the resource section of allowed questions

The screenshot shows a testing environment interface. At the top, there are two input fields for optimization speedup. The first field is labeled 'Optimization A speedup' and contains the text 'number (rtol=0.01, atol=1e-08)'. The second field is labeled 'Optimization B speedup' and also contains the text 'number (rtol=0.01, atol=1e-08)'. Below these fields is a section titled 'Resources (Calculator, Python Console, x86 quick ref, etc.)'. This section contains a message: 'This is a sandboxed Python console with some helpful functions for dealing with doing binary arithmetic. You can use it as you like.' Below this message is a yellow warning box that says: 'Warning: This is in beta. It may or may not be available on the final in the CBTF. It's not clear if the machines there can handle it. If you have problems or suggestions, please post on Piazza.'

Computer Science & Engineering

203

つづく

