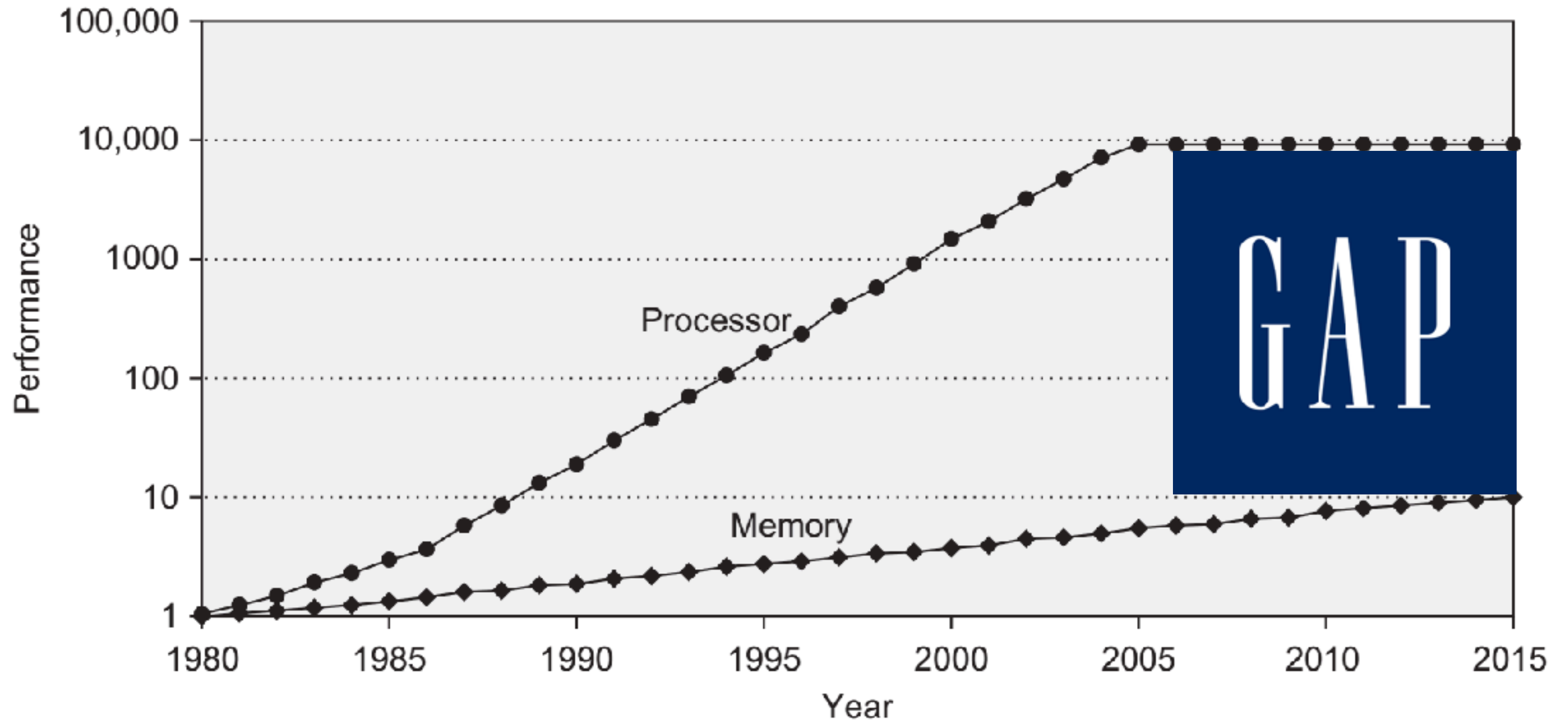


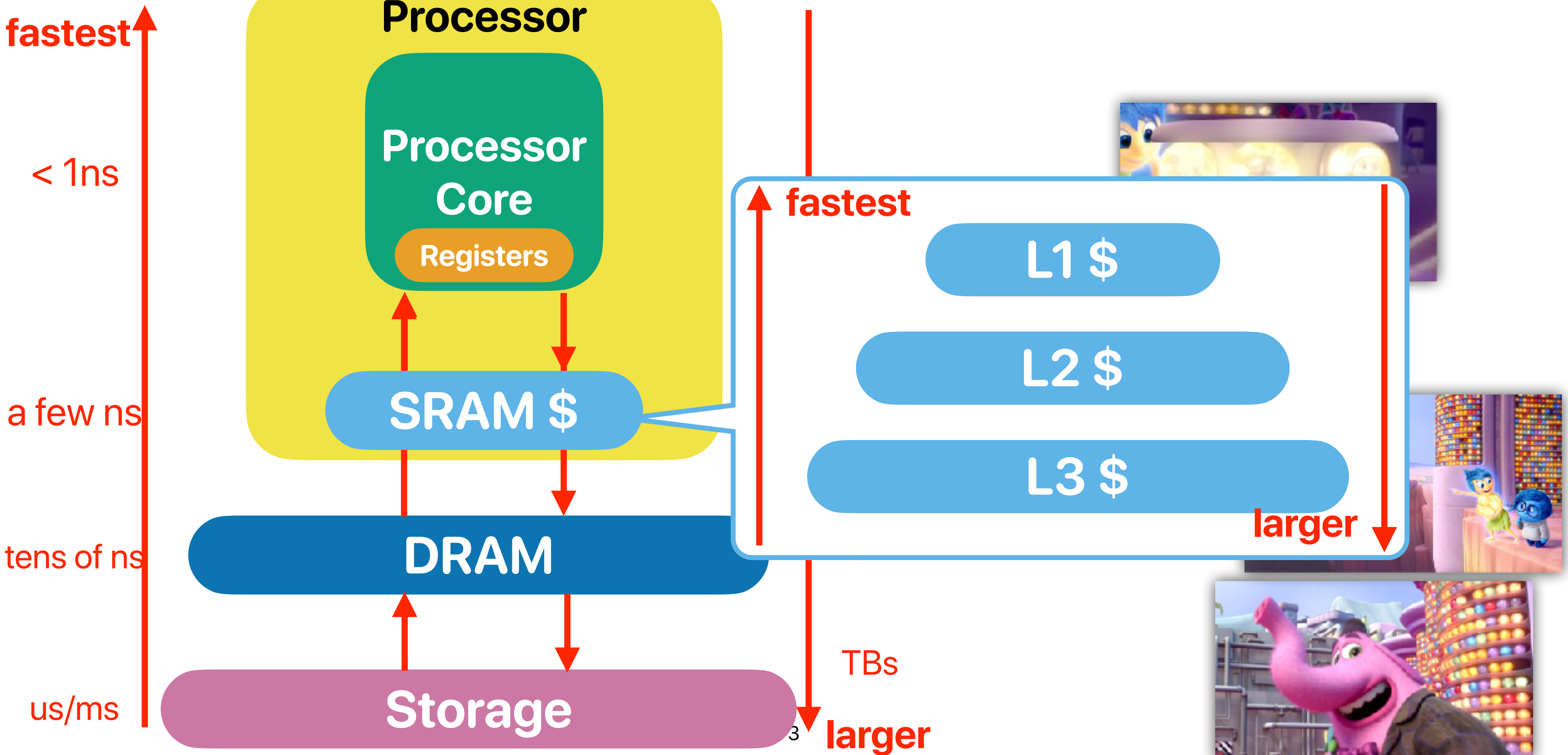
Memory Hierarchy (4): Causes & remedies to cache misses — the hard way

Hung-Wei Tseng

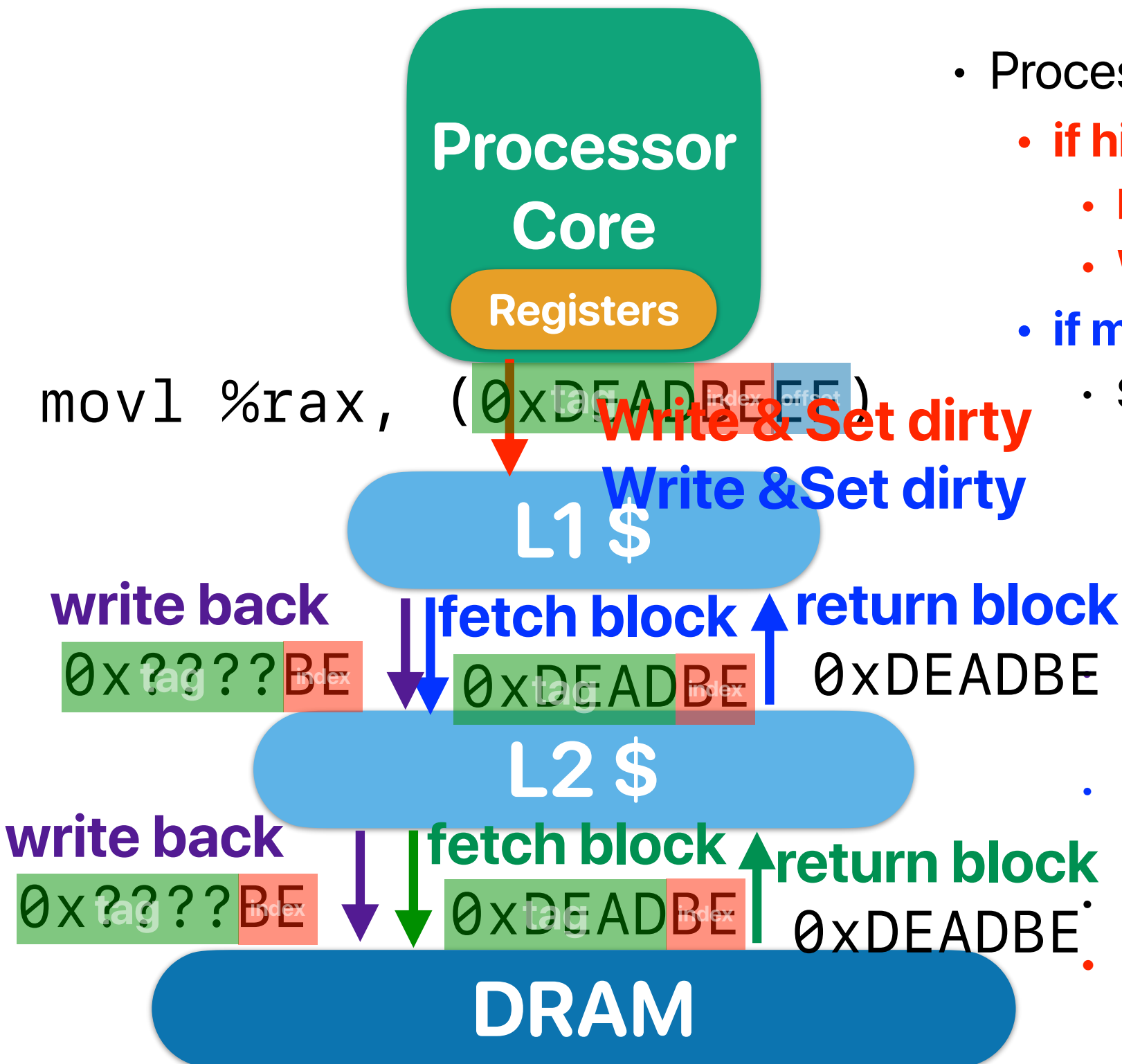
Recap: Performance gap between Processor/Memory



Memory Hierarchy



The complete picture



- Processor sends memory access request to L1-\$
 - **if hit**
 - **Read - return data**
 - **Write - update & set DIRTY**
 - **if miss**
 - Select a victim block
 - If the target "set" is not full — select an empty/invalidated block as the victim block
 - If the target "set" is full — select a victim block using some policy
 - LRU is preferred — to exploit temporal locality!
 - If the victim block is "dirty" & "valid"
 - **Write back** the block to lower-level memory hierarchy
 - Fetch the requesting block from lower-level memory hierarchy and place in the victim block
 - If write-back or fetching causes any miss, repeat the same process
 - **Present the write "ONLY" in L1 and set DIRTY**

Review: C = ABS

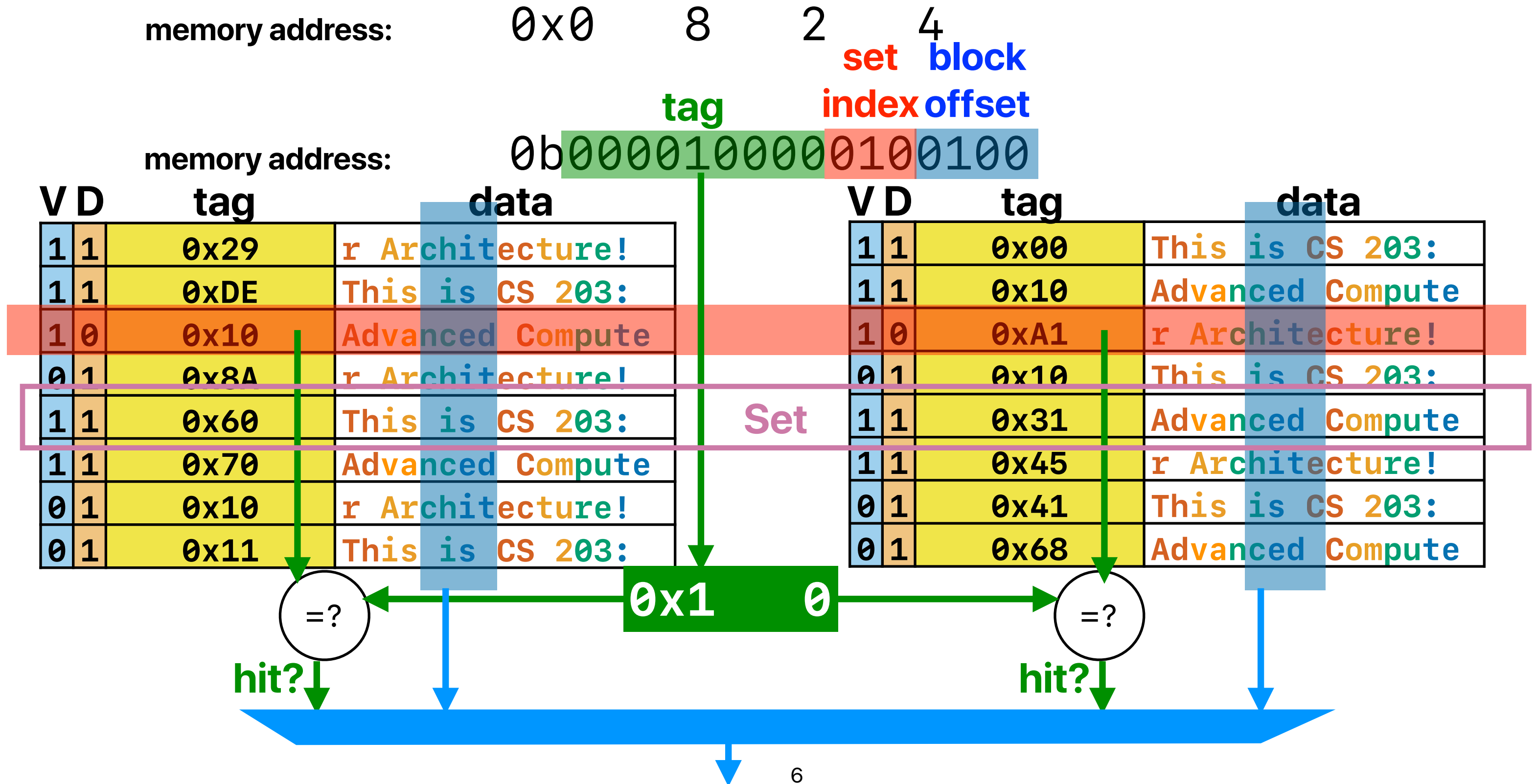
- **C**: Capacity in data arrays
- **A**: Way-**A**ssociativity — how many blocks within a set
 - N-way: N blocks in a set, A = N
 - 1 for direct-mapped cache
- **B**: Block Size (Cacheline)
 - How many bytes in a block
- **S**: Number of **S**ets:
 - A set contains blocks sharing the same index
 - 1 for fully associate cache
- number of bits in **b**lock offset — $\lg(\mathbf{B})$
- number of bits in **s**et index: $\lg(\mathbf{S})$
- tag bits: $\text{address_length} - \lg(\mathbf{S}) - \lg(\mathbf{B})$
 - address_length is 64 bits for 64-bit machine
- $\frac{\text{address}}{\text{block_size}} \pmod{S} = \text{set index}$

memory address:

0b 000010000 010 0100

tag set block
index offset

Way-associative cache



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

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[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
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?
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]
⋮	⋮	⋮	⋮	⋮	⋮	⋮

intel Core i7

- 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
32KB = 8 * 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[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 (cont.)

- Size 32KB, 8-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
32KB = 8 * 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	Miss	
b[8]	0x20040	0b00100000000001000000	0x20	0x1	Miss	
c[8]	0x30040	0b00110000000001000000	0x30	0x1	Miss	
d[8]	0x40040	0b01000000000001000000	0x40	0x1	Miss	
e[8]	0x50040	0b01010000000001000000	0x50	0x1	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	

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

Miss when the array index is a multiply of 8!

Outline

- 3Cs — where cache misses are coming from
- Optimizing cache performance — the hardware perspective

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



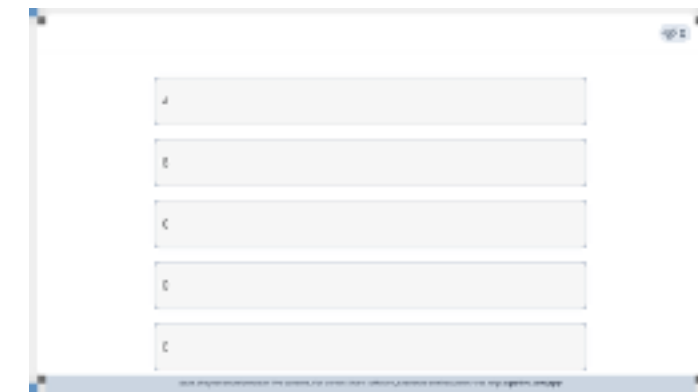
NVIDIA Tegra X1

- 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[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]
}
```

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[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
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?
a[0]	0x10000	0 b0001000 000000000000000	0x8	0x0	Compulsory Miss	
b[0]	0x20000	0 b0010000 000000000000000	0x10	0x0	Compulsory Miss	
c[0]	0x30000	0 b0011000 000000000000000	0x18	0x0	Compulsory Miss	
d[0]	0x40000	0 b0100000 000000000000000	0x20	0x0	Compulsory Miss	
e[0]	0x50000	0 b0101000 000000000000000	0x28	0x0	Compulsory Miss	a[0-7]
a[1]	0x10008	0 b0001000 00000000001000	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]
⋮	⋮	⋮	⋮	⋮	⋮	⋮



intel Core i7

- D-L1 Cache configuration of intel Core i7
 - Size 32KB, 8-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]
}
```

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

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

A screenshot of a Pollev poll interface. It shows a list of five empty input boxes, each preceded by a letter (A, B, C, D, E), corresponding to the multiple-choice options. The interface is clean and modern, with a light blue header and a white background for the poll area.

intel Core i7

- Size 32KB, 8-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
32KB = 8 * 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[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 (cont.)

- Size 32KB, 8-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
32KB = 8 * 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

- D-L1 Cache configuration of intel Core i7
 - Size 32KB, 8-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]
}
```

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

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

Basic Hardware Optimization in Improving 3Cs



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 Increasing associativity can reduce conflict misses
- ② Increasing associativity can reduce hit time
- ③ Increasing block size can increase the miss penalty
- ④ Increasing block size can reduce compulsory misses

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 open-ended poll where users enter their answers.

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 Increasing associativity can reduce conflict misses
- ② Increasing associativity can reduce hit time
- ③ Increasing block size can increase the miss penalty
- ④ Increasing block size can reduce compulsory misses

A. 0

B. 1

C. 2

D. 3

E. 4

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

You bring more into the cache
when a miss occurs

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 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

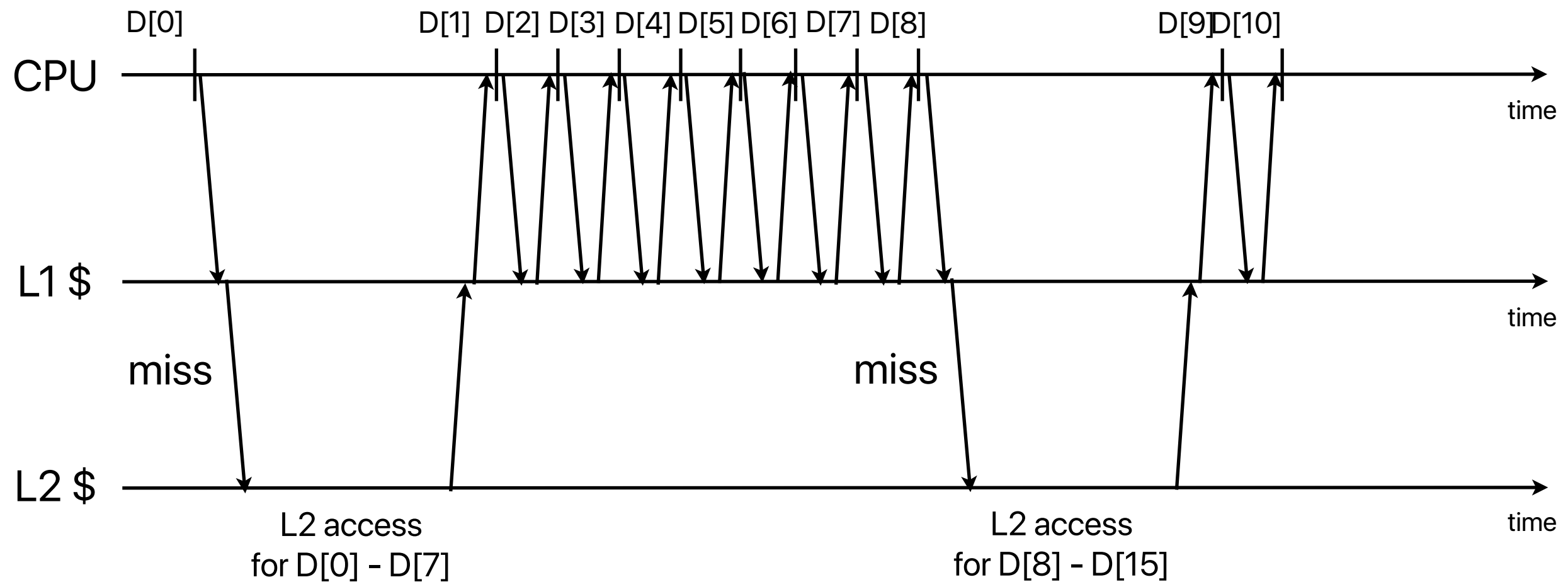
```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
}
```



Prefetching

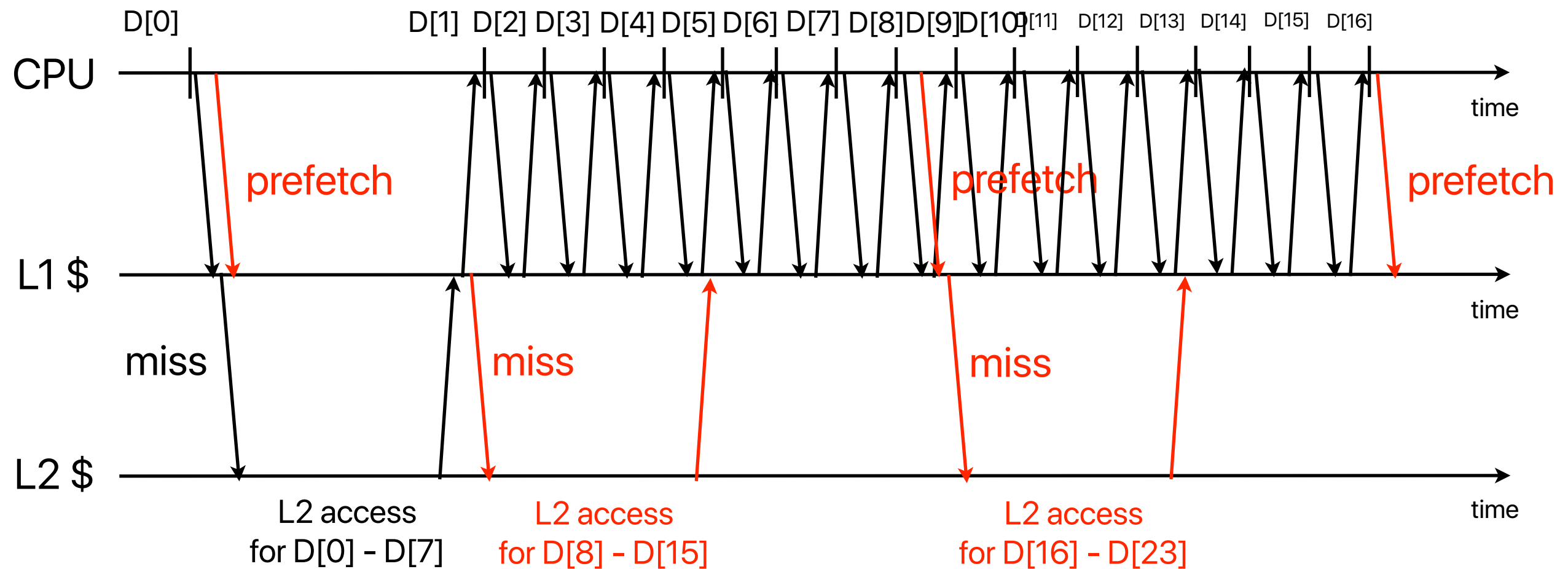
Characteristic of memory accesses

```
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  
}
```



Prefetching

- 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
- Hardware prefetch
 - The processor can keep track the distance between misses. If there is a pattern, fetch $\text{miss_data_address} + \text{distance}$ for a miss
- Software prefetch
 - Load data into X0
 - Using prefetch instructions

Demo

- 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



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

(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

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

(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

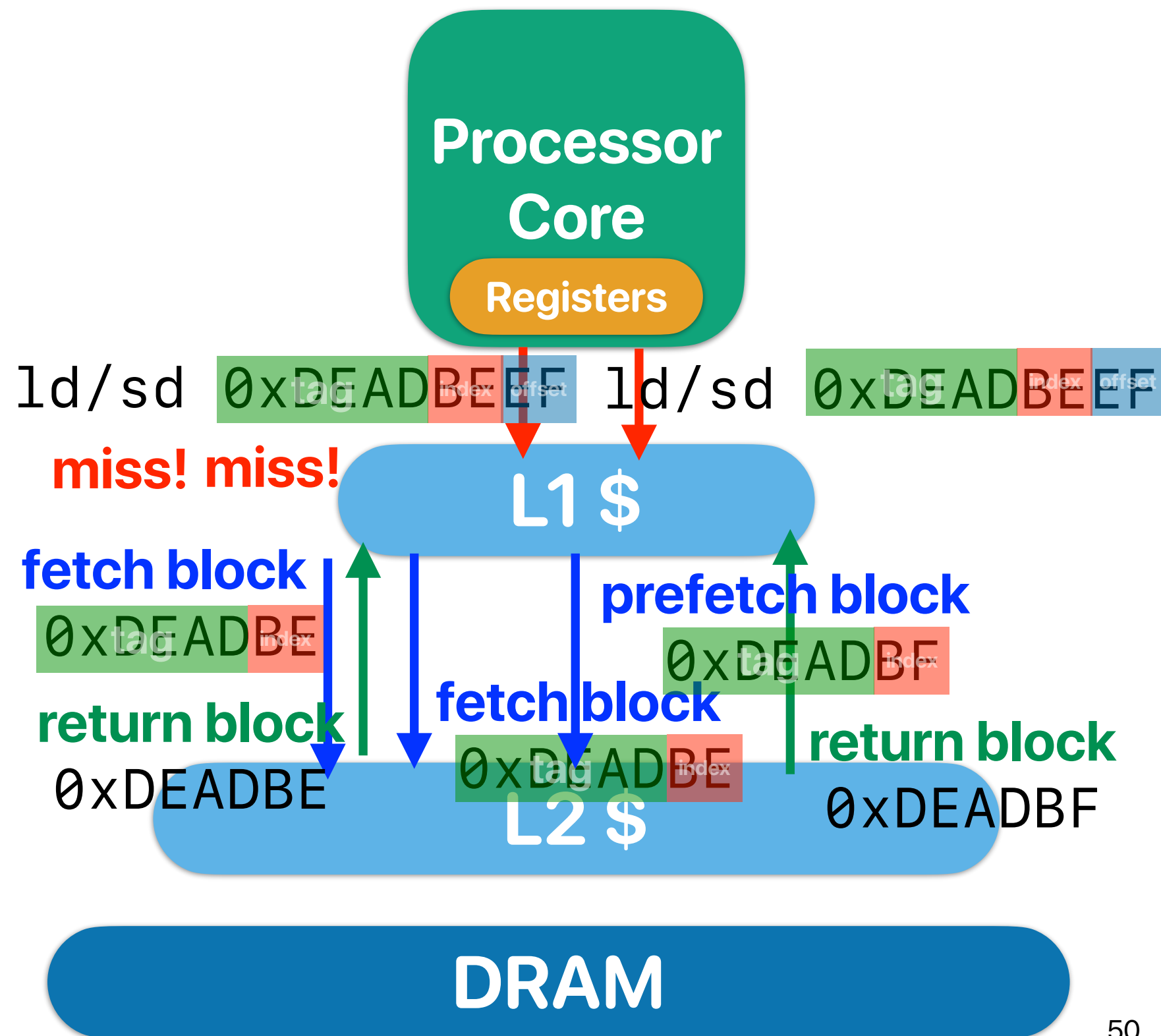
B. 1

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.

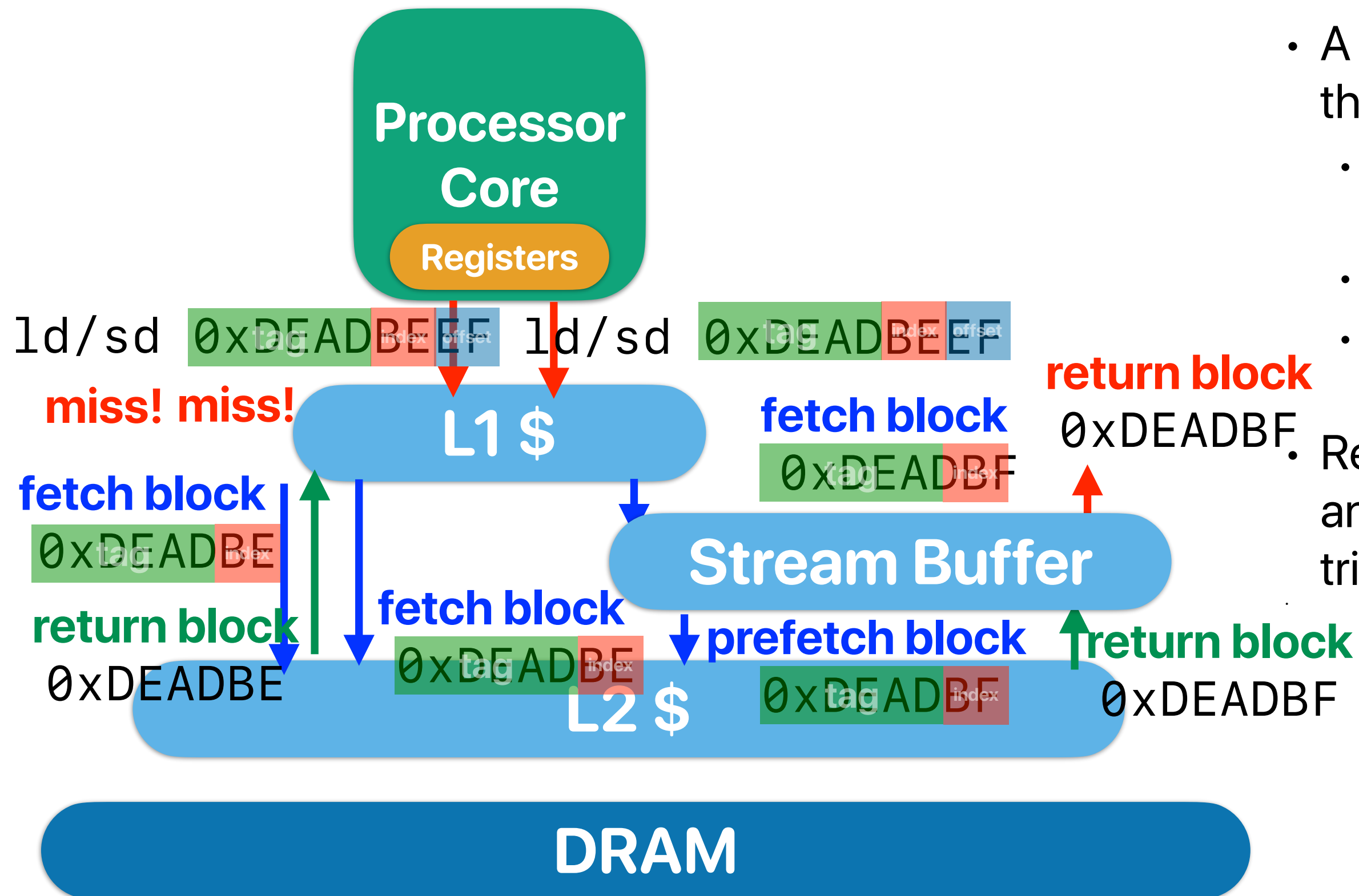
```
double a[8192], b[8192], c[8192], d[8192], e[8192];
/* a = 0x10000, b = 0x20000, c = 0x30000, d = 0x40000, e = 0x50000 */
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
    //load a[i], b[i], c[i], d[i] and then store to e[i]
```

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

	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]	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	

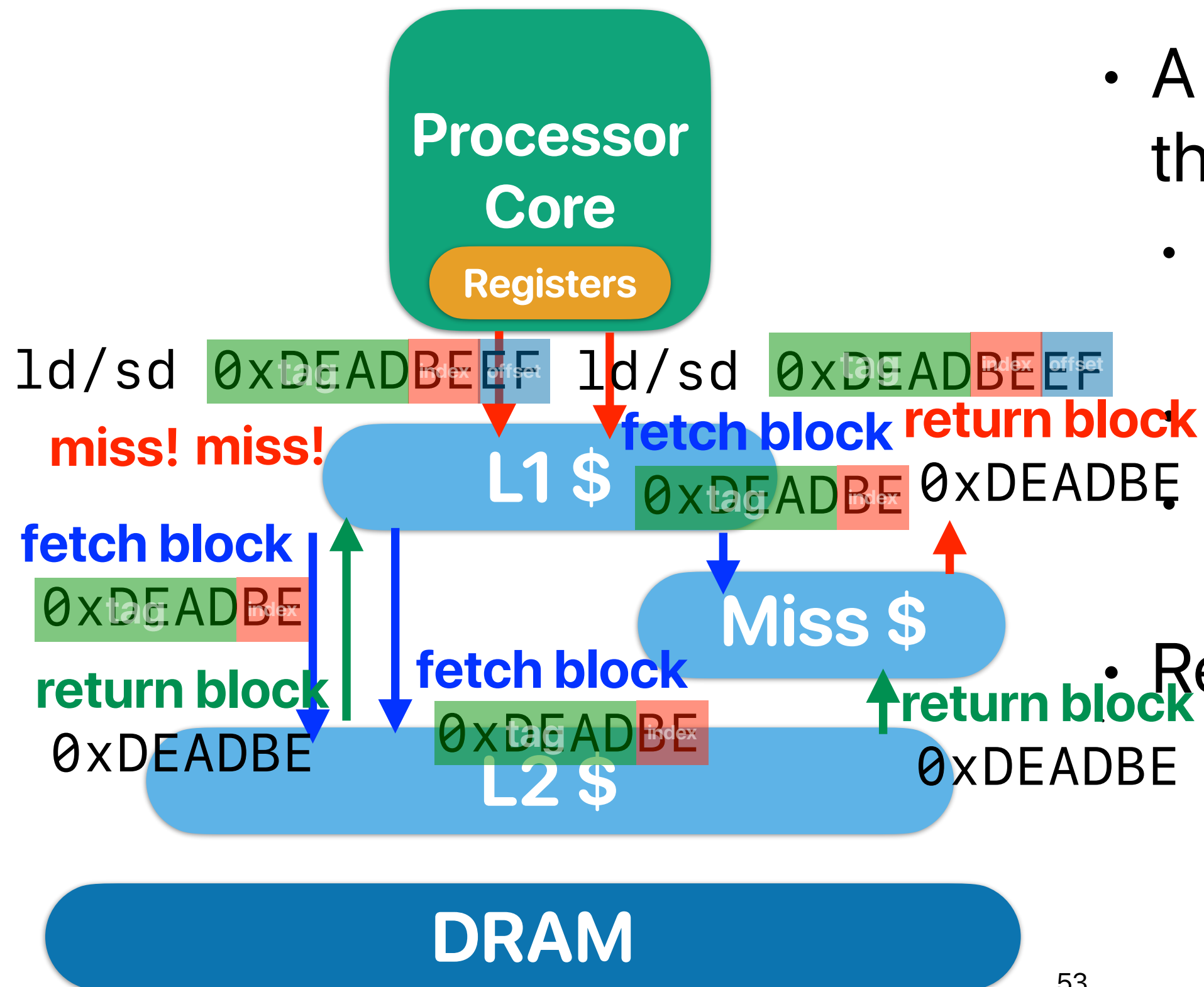
100% miss rate!

Stream buffer



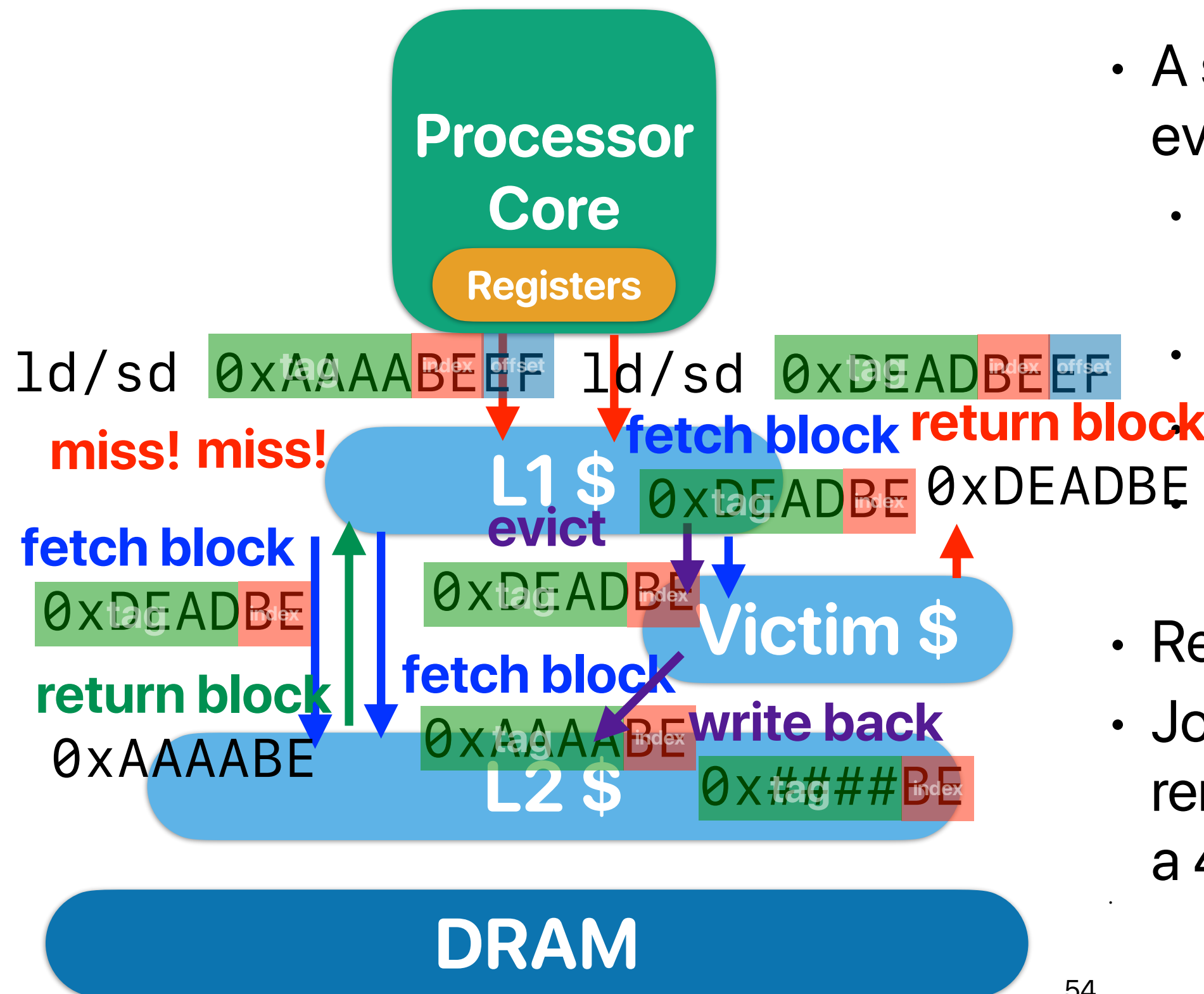
- A small cache that captures the prefetched blocks
 - Can be built as fully associative since it's small
 - Consult when there is a miss
 - Retrieve the block if found in the stream buffer
- Reduce compulsory misses and avoid conflict misses triggered by prefetching

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

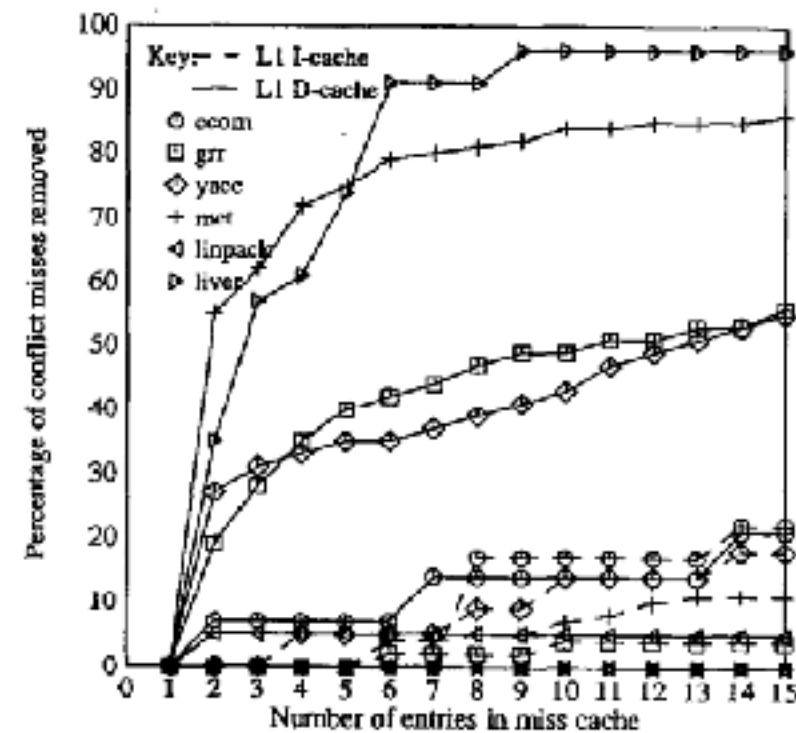


Figure 3-3: Conflict misses removed by miss caching

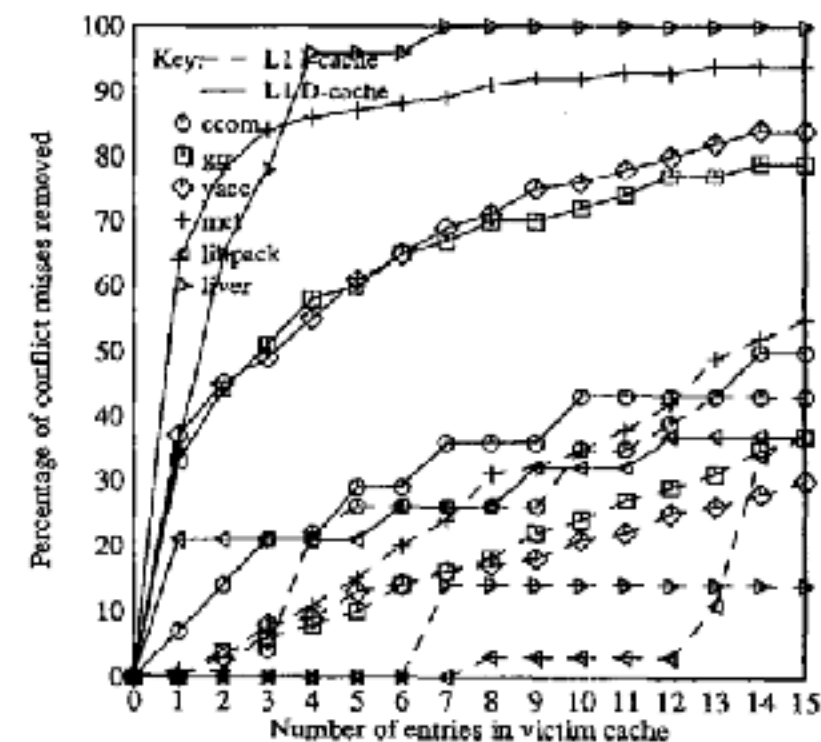


Figure 3-5: Conflict misses removed by victim caching

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

Announcement

- Reading quiz #4 due **next Tuesday** before the lecture
- Assignment #3 is released, due **next Thursday**
 - Please check our course website
<https://www.escalab.org/classes/cs203-2024sp/>
 - Start early to get feedback from the autograder
- Hung-Wei's office hour this week moves to Thursday 2p-4p
- Regarding autograder
 - It's not an AI — don't be too creative
 - Please make sure your answer aligned with the representation we've in the assignment examples and lectures
 - Please be as detailed as possible — we do word count, we do want to find key words within your answers — you will need to do the same for midterm. Don't just show the final result
 - Please make sure you include the exact numbers you measured
 - Copy-paste from your measurements is typically a good strategy

Computer Science & Engineering

203

つづく

