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

Hung-Wei Tseng

von Neumman Architecture

509cbd23

00c2e800

Processor

Instructions	Data
0f00bb27	00c2e800
509cbd23	00000008
00005d24	00c2f000
0000bd24	00000008
2ca422a0	00c2f800
130020e4	00000008
00003d24	00c30000
2ca4e2b3	00000008

Memory

Program

Instructions	Data
0f00bb27	00c2e800
509cbd23	00000008
00005d24	00c2f000
0000bd24	00000008
2ca422a0	00c2f800
130020e4	00000008
00003d24	00c30000
2ca4e2b3	00000008

Storage

Recap: Memory Hierarchy

fastest

< 1ns

a few ns

tens of ns

us/ms

Processor

Processor Core

Registers

SRAM $

DRAM

Storage

TBs

larger

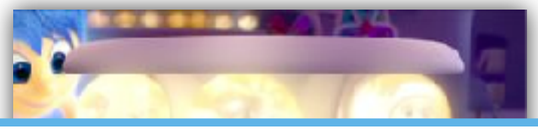
fastest

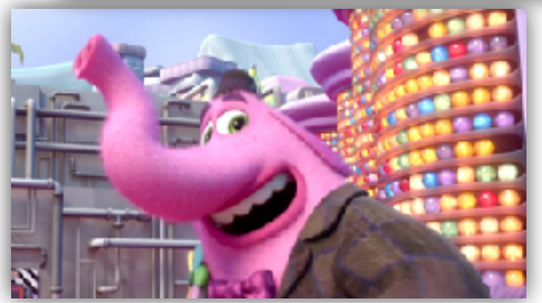
L1 $

L2 $

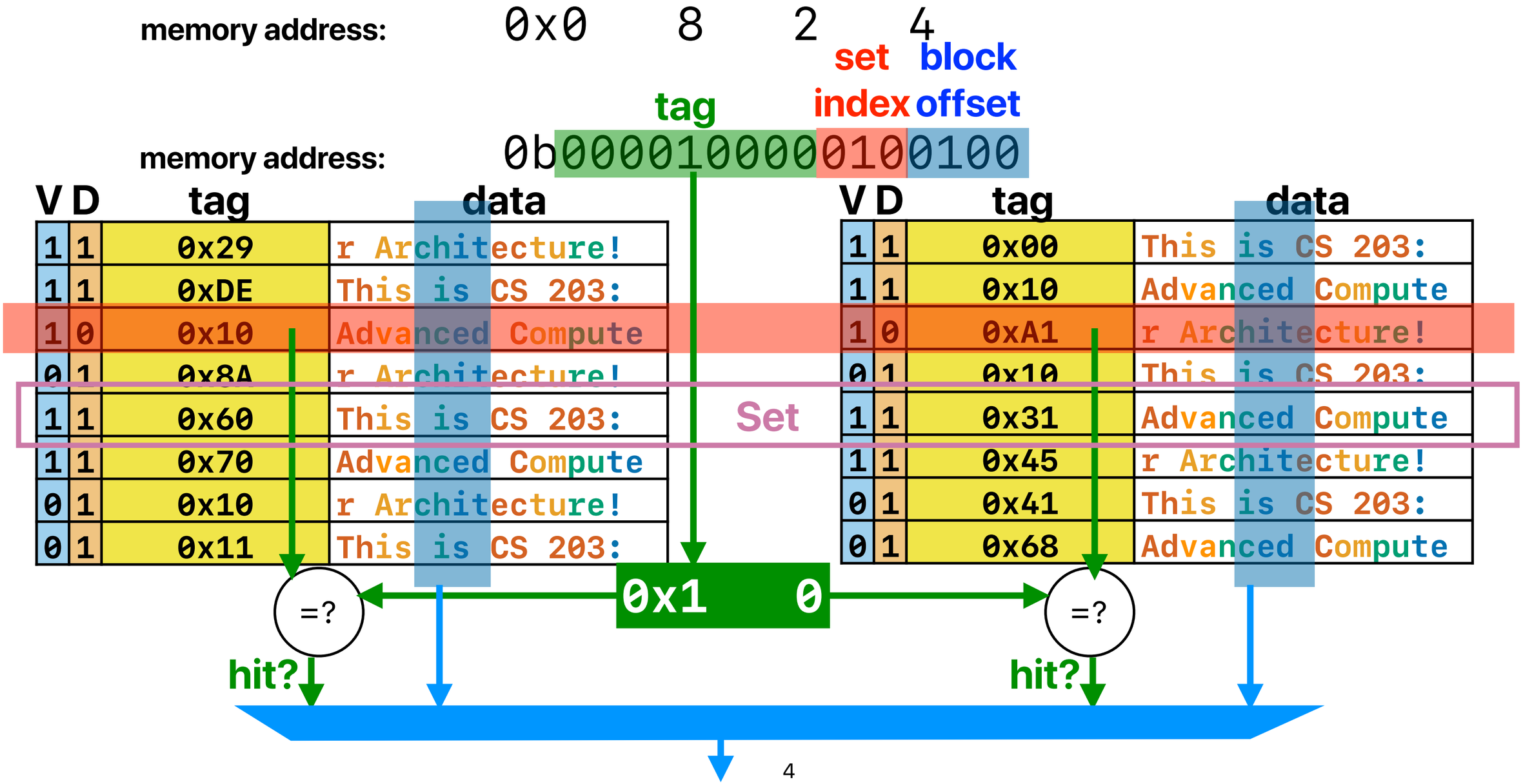
L3 $

larger





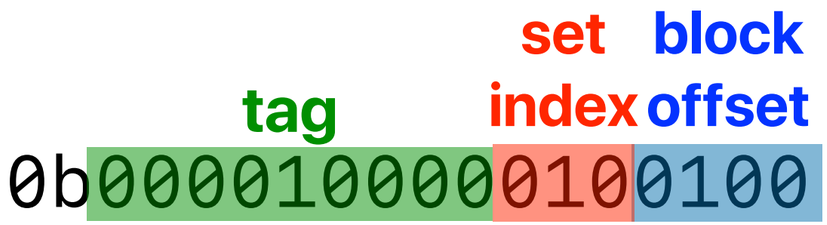
Recap: Way-associative cache



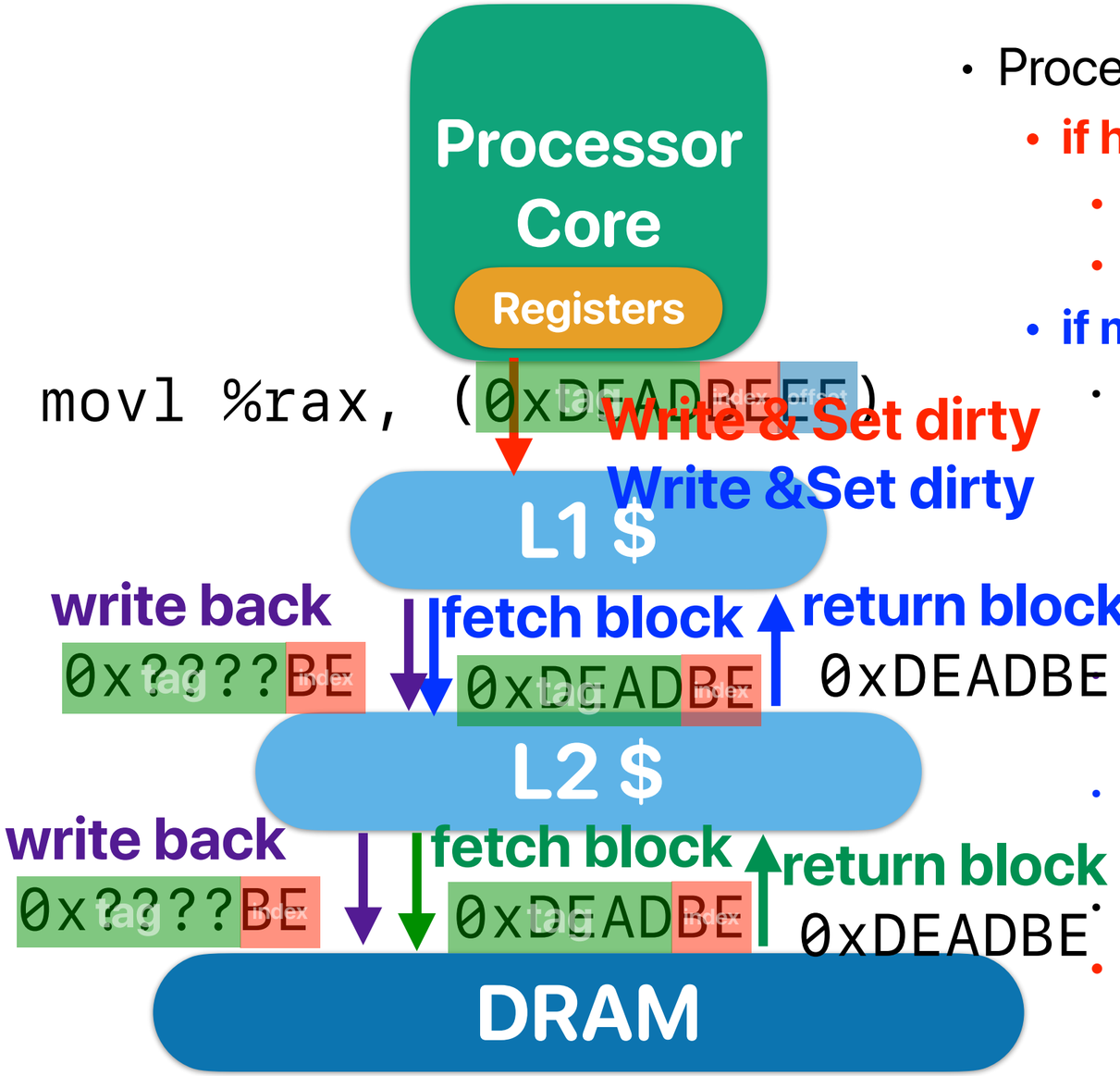
Review: C = ABS

- **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:



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**

Outline

- 3Cs of cache misses
- Software optimizations for cache performance

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

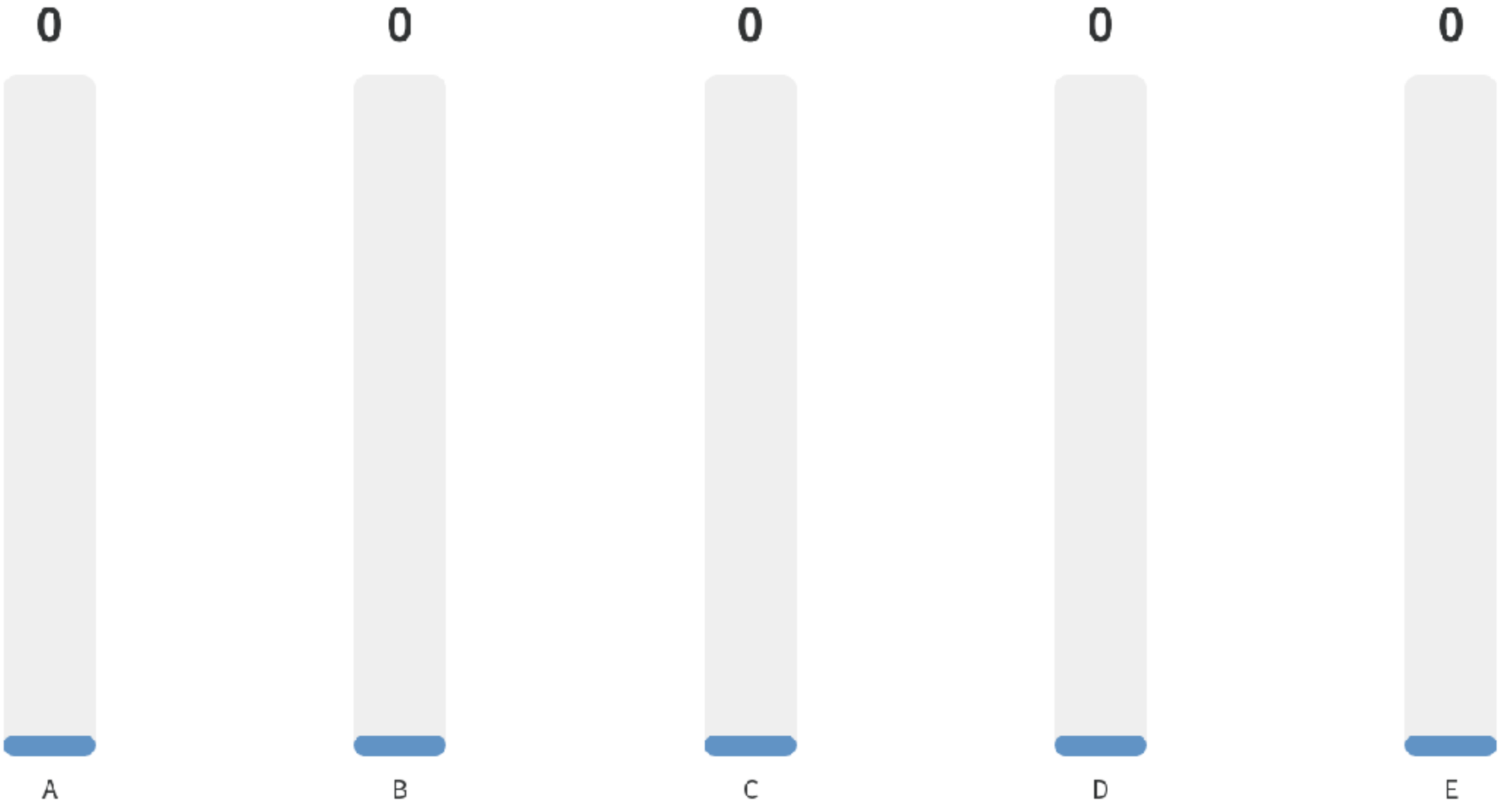
- D-L1 Cache configuration of NVIDIA Tegra X1
  - Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
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%

0



NVIDIA Tegra X1

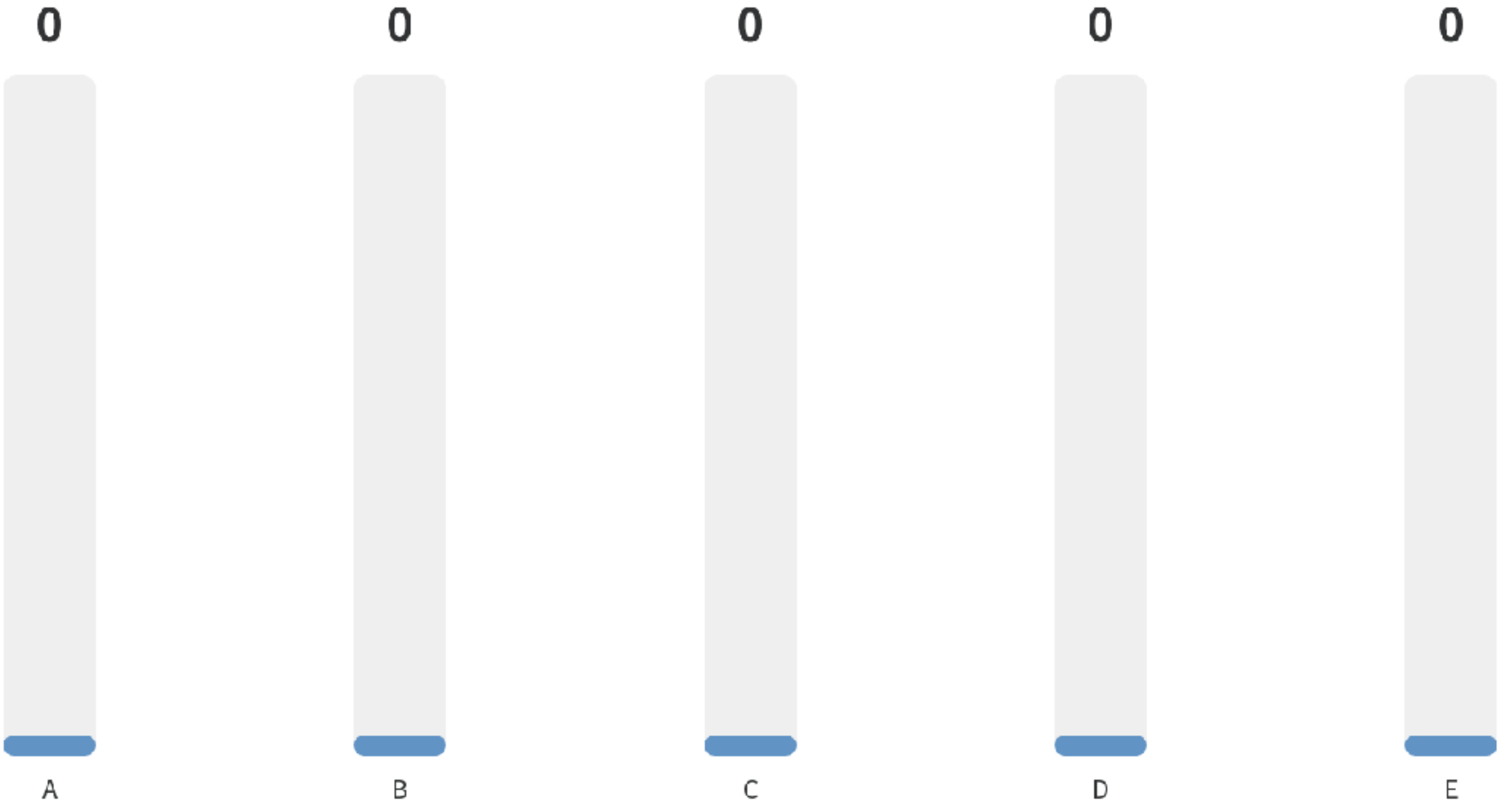
- D-L1 Cache configuration of NVIDIA Tegra X1
  - Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
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%

0



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

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]
⋮	⋮	⋮	⋮	⋮	⋮	⋮

intel Core i7

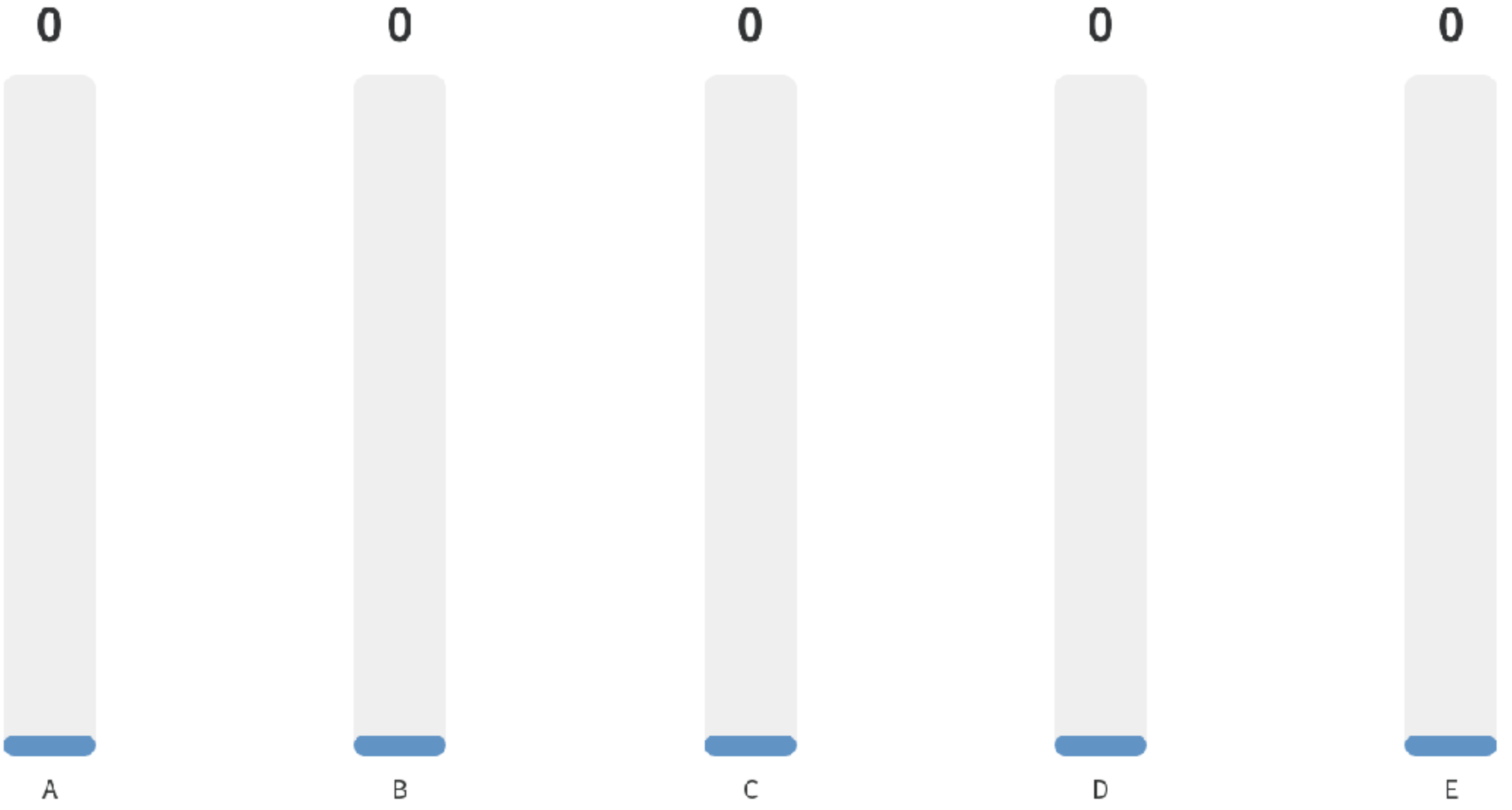
- D-L1 Cache configuration of intel Core i7
  - Size 32KB, 8-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
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%

0



intel Core i7

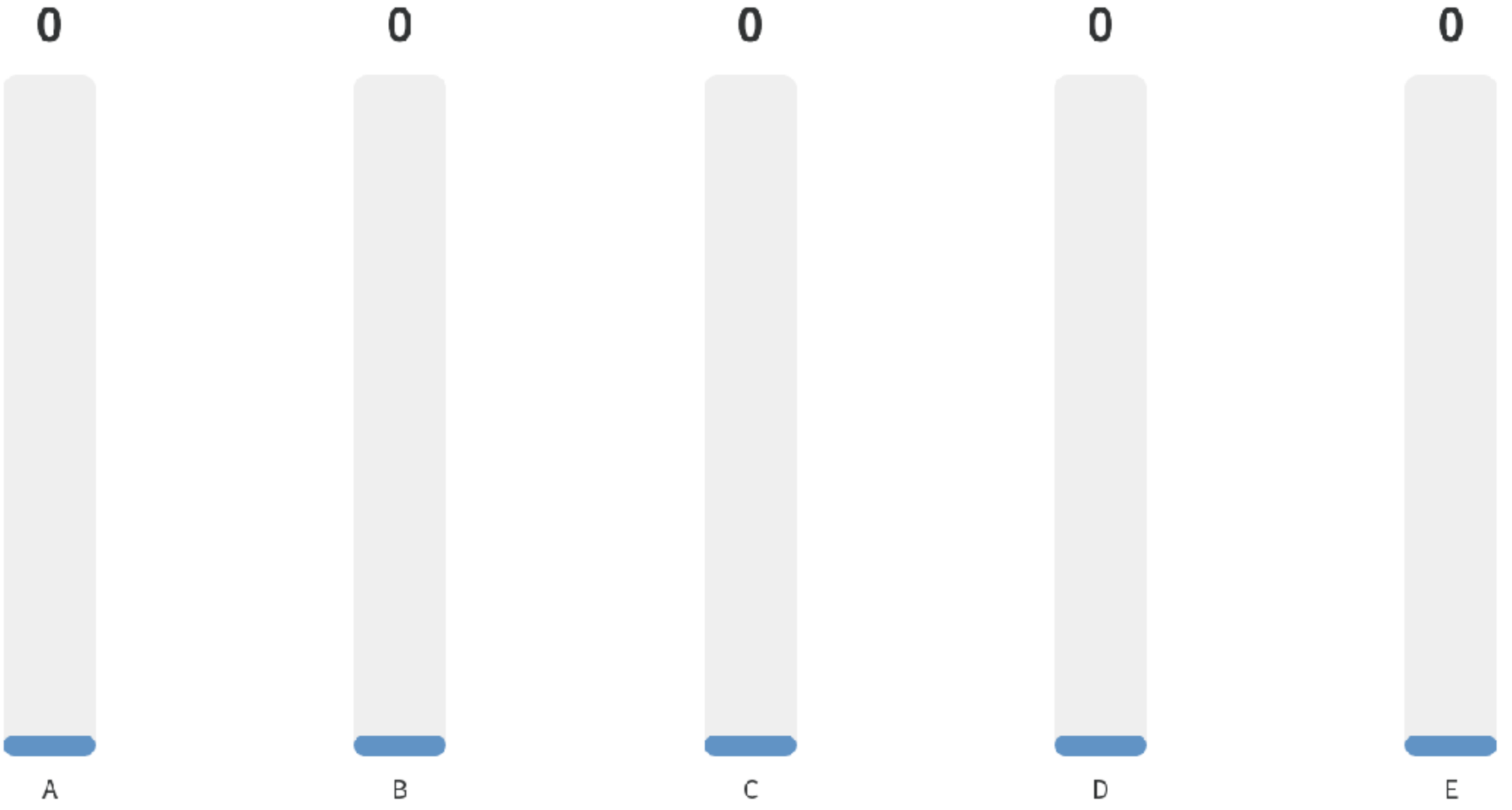
- D-L1 Cache configuration of intel Core i7
  - Size 32KB, 8-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
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%

0



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

tag index offset

	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

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 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?

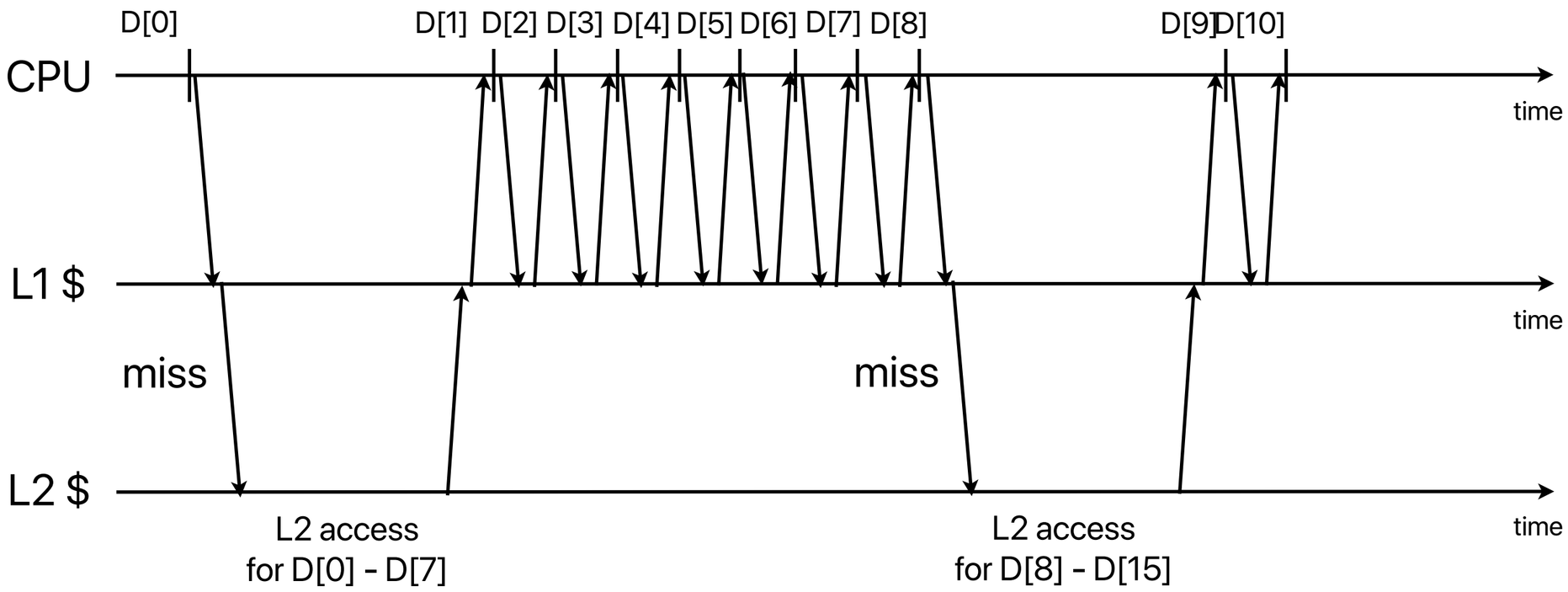
A. 12.5%
B. 66.67%
C. 68.75%
D. 87.5%
E. 100%

How can programmer improve memory performance?

Prefetching

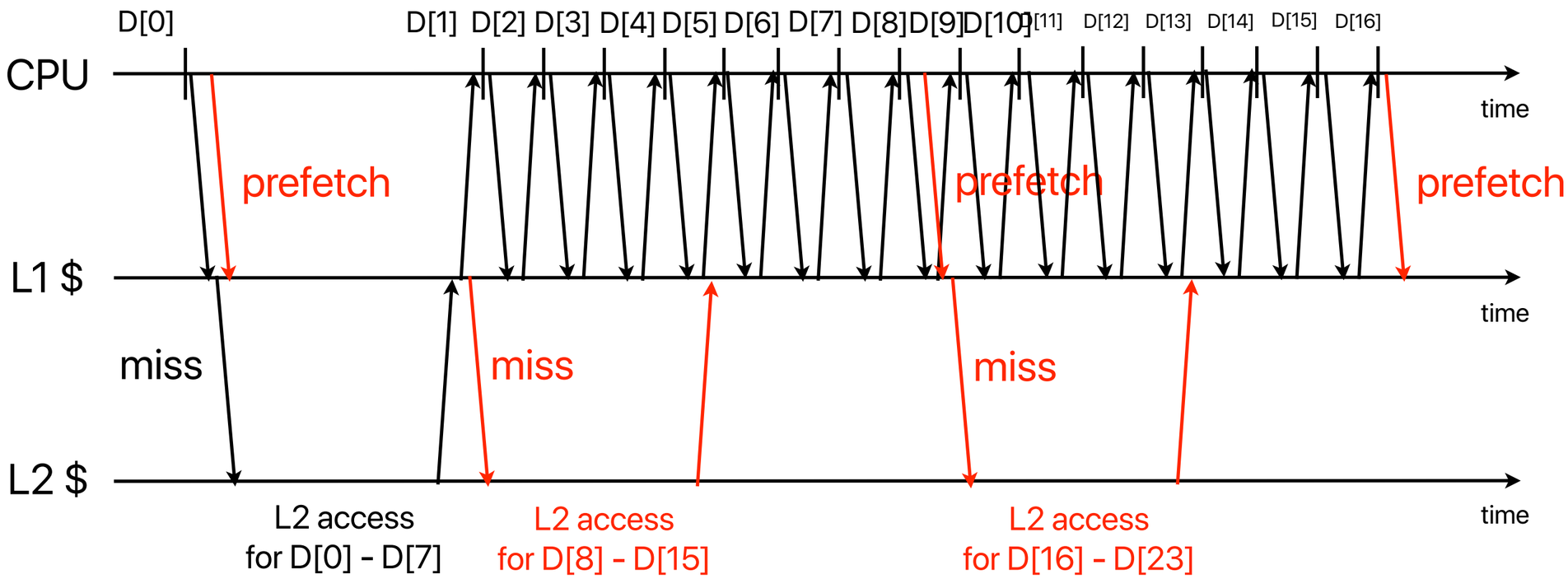
Characteristic of memory accesses

```
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 miss_data_address+distance for a miss
- Software prefetch
  - Load data into some register
  - Using prefetch instructions

Data structures

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

The result of `sizeof(struct student)`

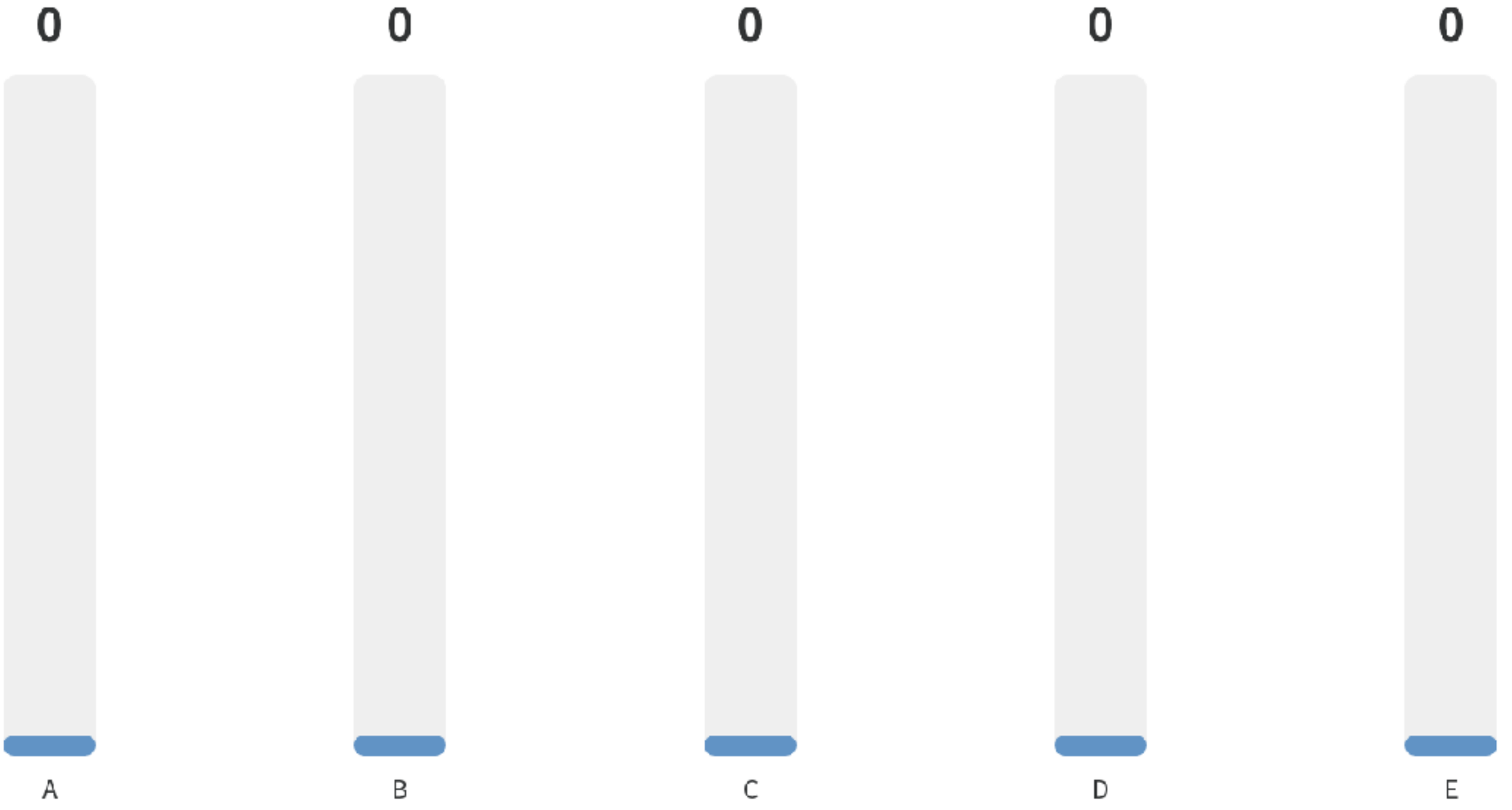
- Consider the following data structure:

```
struct student {
    int id;
    double *homework;
    int participation;
    double midterm;
    double average;
};
```

What's the output of `printf("%lu\n",sizeof(struct student))`?

A. 20
B. 28
C. 32
D. 36
E. 40

0



The result of `sizeof(struct student)`

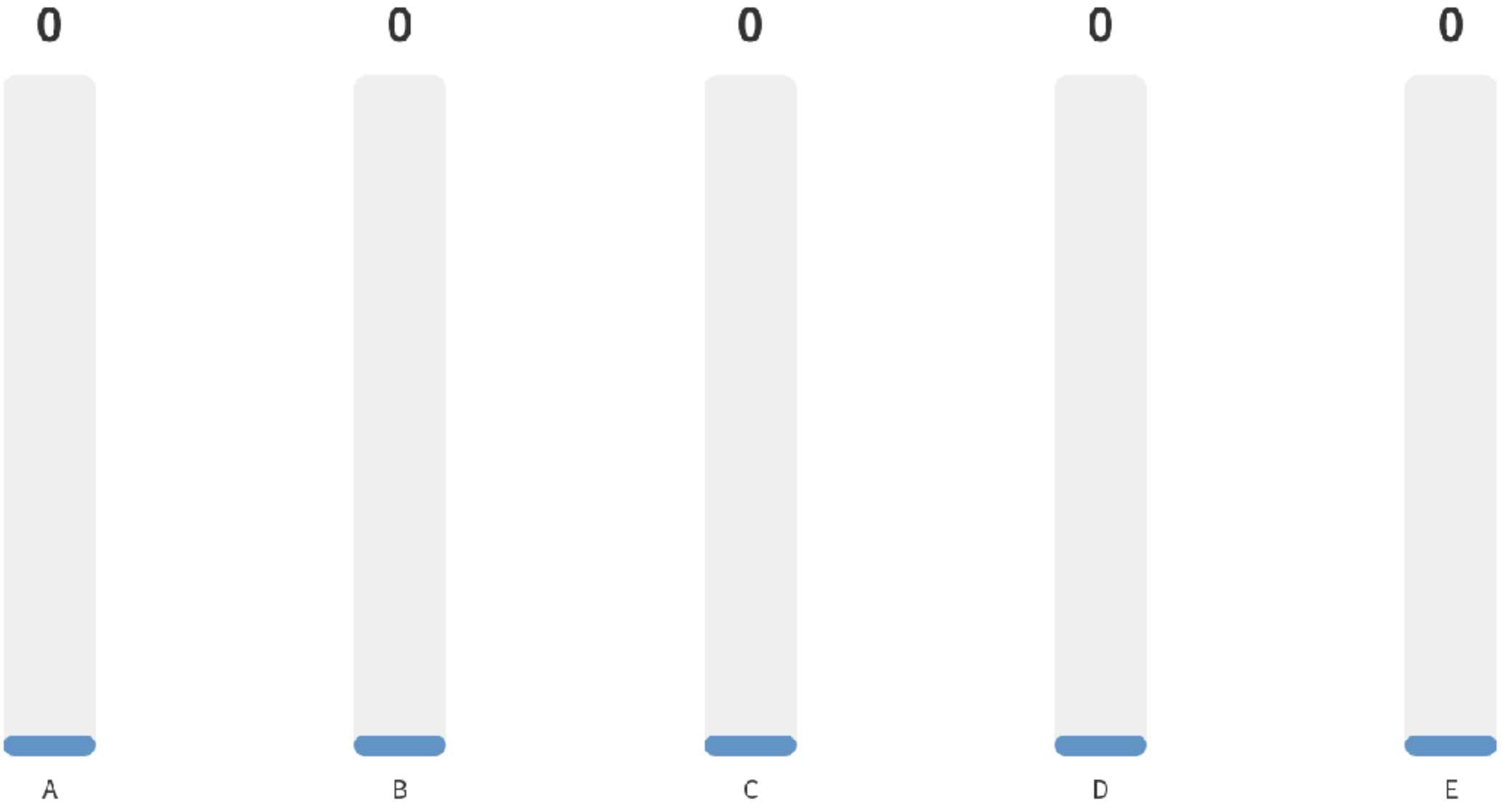
- Consider the following data structure:

```
struct student {
    int id;
    double *homework;
    int participation;
    double midterm;
    double average;
};
```

What's the output of `printf("%lu\n",sizeof(struct student))`?

A. 20
B. 28
C. 32
D. 36
E. 40

0



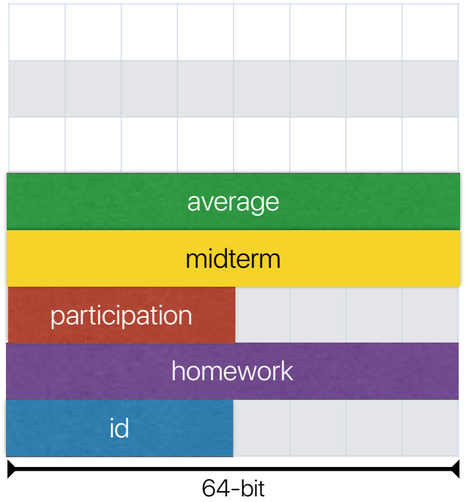
Memory addressing/alignment

- Almost every popular ISA architecture uses "byte-addressing" to access memory locations
- Instructions generally work faster when the given memory address is aligned
  - Aligned — if an instruction accesses an object of size n at address X, the access is **aligned** if $X \bmod n = 0$.
  - Some architecture/processor does not support aligned access at all
  - Therefore, compilers only allocate objects on "aligned" address

The result of `sizeof(struct student)`

- Consider the following data structure:

```
struct student {
    int id;
    double *homework;
    int participation;
    double midterm;
    double average;
};
```



What's the output of `printf("%lu\n",sizeof(struct student))`?

A. 20
B. 28
C. 32
D. 36
E. 40

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

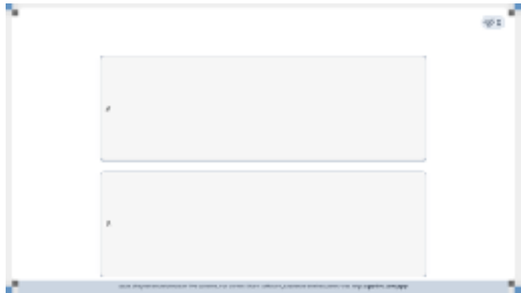
Column-store or row-store

- Considering your the most frequently used queries in your database system are similar to
  `SELECT AVG(assignment_1) FROM table`
  Which of the following would be a data structure that better implements the table supporting this type of queries?

Array of objects	object of arrays

`struct grades`
`{`
`  int id;`
`  double *homework;`
`  double average;`
`};`
`table = (struct grades *) \`
`malloc(num_of_students*sizeof(struct grades));`
	`struct grades`
	`{`
	`  int *id;`
	`  double **homework;`
	`  double *average;`
	`};`
	
	`table =(struct grades *)malloc(sizeof(struct grades));`

A. Array of objects
B. Object of arrays



0

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

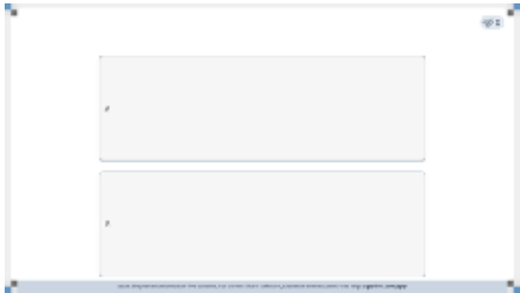
Column-store or row-store

- Considering your the most frequently used queries in your database system are similar to
  `SELECT AVG(assignment_1) FROM table`
  Which of the following would be a data structure that better implements the table supporting this type of queries?

Array of objects	object of arrays

`struct grades`
`{`
`  int id;`
`  double *homework;`
`  double average;`
`};`
`table = (struct grades *) \`
`malloc(num_of_students*sizeof(struct grades));`
	`struct grades`
	`{`
	`  int *id;`
	`  double **homework;`
	`  double *average;`
	`};`
	
	`table =(struct grades *)malloc(sizeof(struct grades));`

A. Array of objects
B. Object of arrays



0

Column-store or row-store

- Considering your the most frequently used queries in your database system are similar to
  `SELECT AVG(assignment_1) FROM table`
  Which of the following would be a data structure that better implements the table supporting this type of queries?

Array of objects	object of arrays

`struct grades`
`{`
`  int id;`
`  double *homework;`
`  double average;`
`};`
`table = (struct grades *) \`
`malloc(num_of_students*sizeof(struct grades));`
	`struct grades`
	`{`
	`  int *id;`
	`  double **homework;`
	`  double *average;`
	`};`
	
	`table =(struct grades *)malloc(sizeof(struct grades));`

A. Array of objects
B. Object of arrays

What if we want to calculate average scores for each student?

Array of structures or structure of arrays

Array of objects

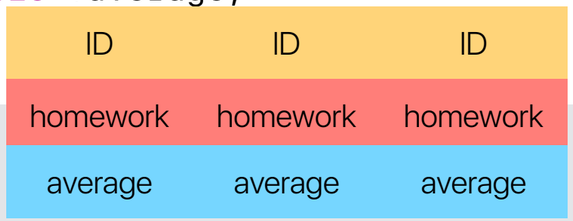
```
struct grades
{
  int id;
  double *homework;
  double average;
};
```

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;
```

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

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]
}
```

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

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

What if the code look like this?

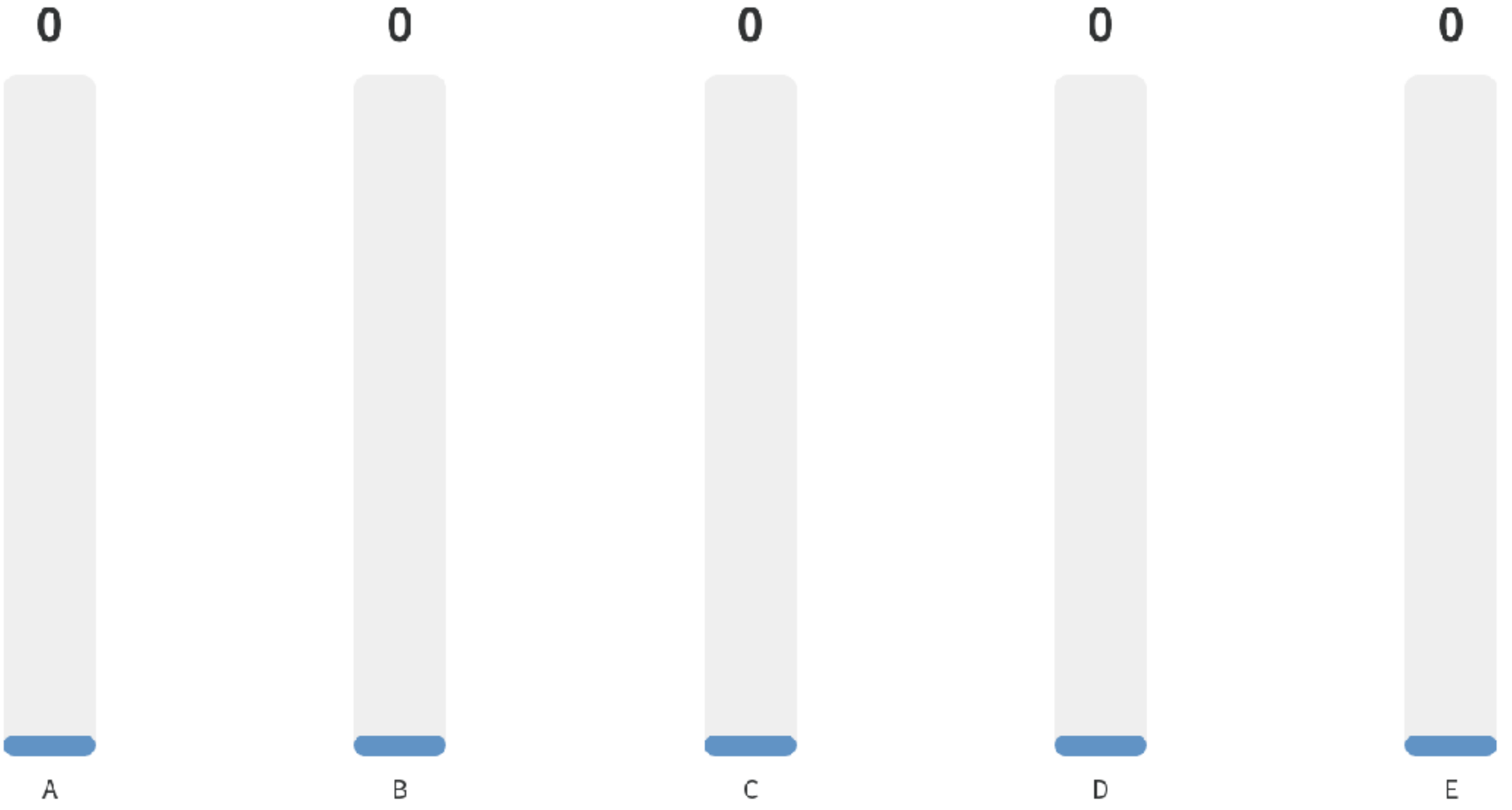
- D-L1 Cache configuration of NVIDIA Tegra X1
  - Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

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

0



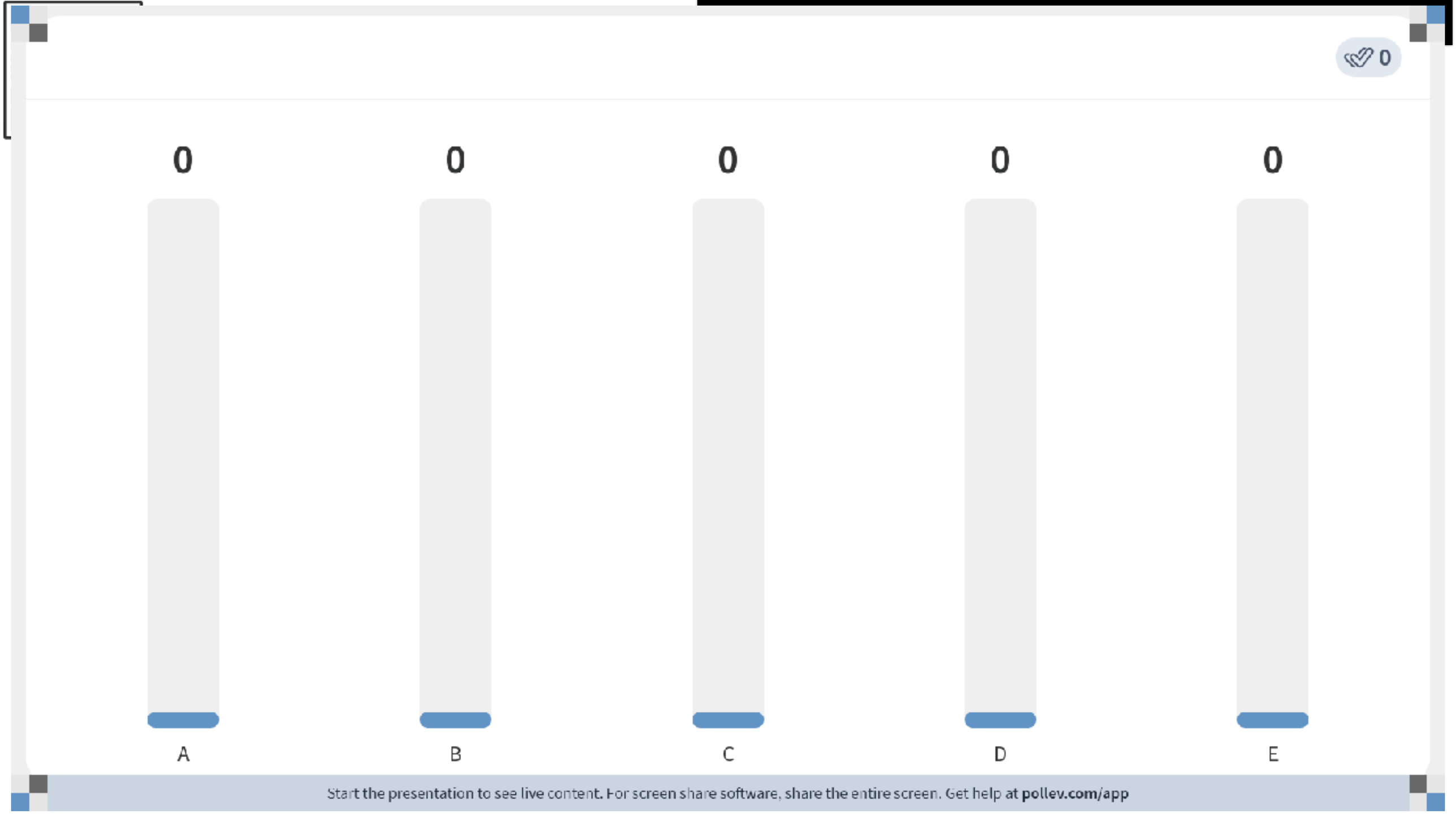
What if the code look like this?

- 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];
/* 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%


0
0
0
0
0
A
B
C
D
E
Start the presentation to see live content. For screen share software, share the entire screen. Get help at pollev.com/app

What if the code look like this?

- 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];
/* 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%

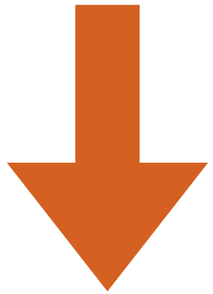
E. 100%

Loop 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];
}
```

Loop fission



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];
```

What if we change the processor?

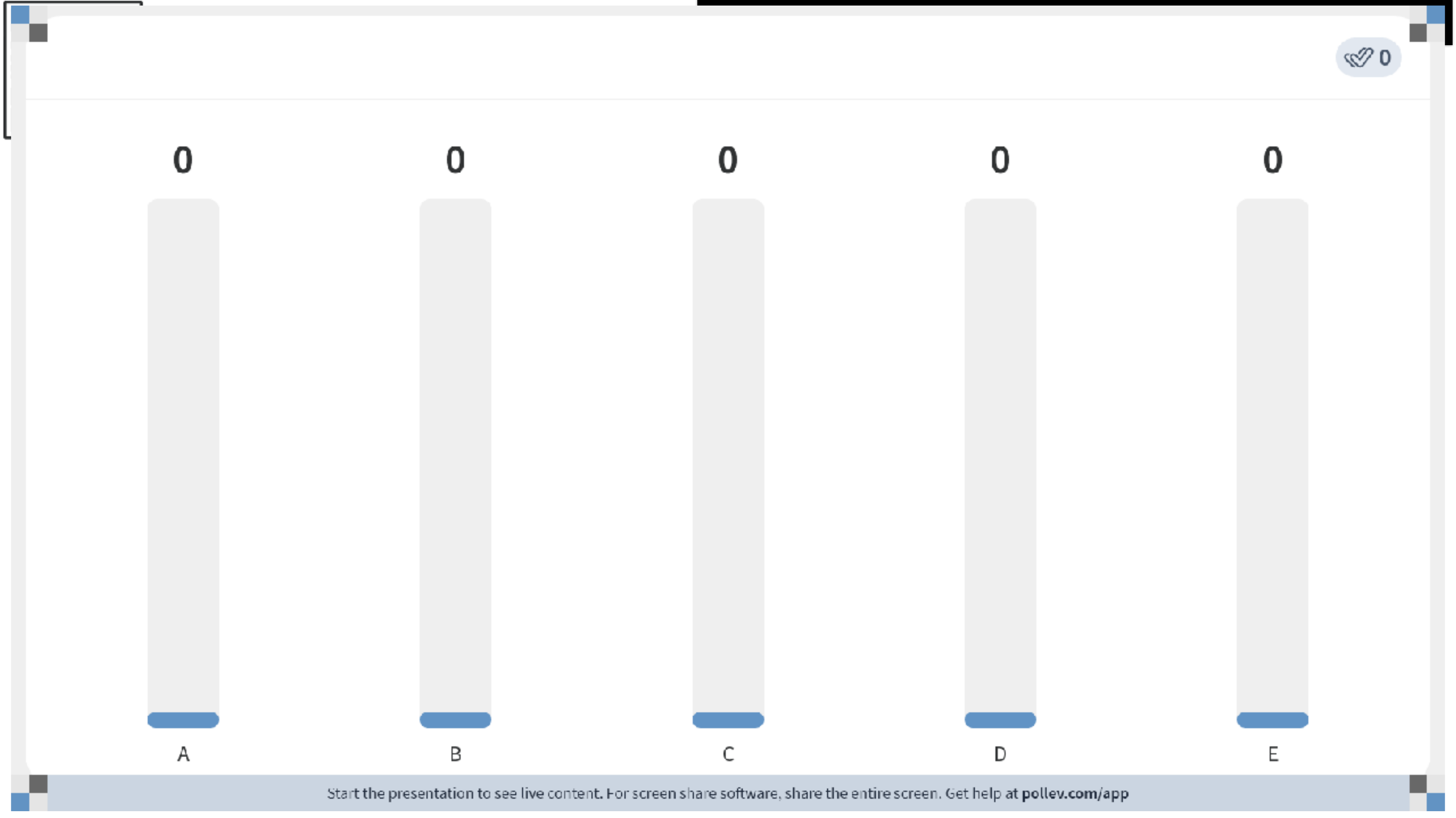
- If we have an intel processor with a 48KB, 12-way, 64B-blocked L1 cache, which version of code performs better?
  A. Version A, because the code incurs fewer cache misses
  B. Version B, because the code incurs fewer cache misses
  C. Version A, because the code incurs fewer memory references
  D. Version B, because the code incurs fewer memory references
  E. They are about the same

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];
}
```


0
0
0
0
0
A
B
C
D
E
Start the presentation to see live content. For screen share software, share the entire screen. Get help at pollev.com/app

What if we change the processor?

- If we have an intel processor with a 48KB, 12-way, 64B-blocked L1 cache, which version of code performs better?
  A. Version A, because the code incurs fewer cache misses
  B. Version B, because the code incurs fewer cache misses
  C. Version A, because the code incurs fewer memory references
  D. Version B, because the code incurs fewer memory references
  E. They are about the same

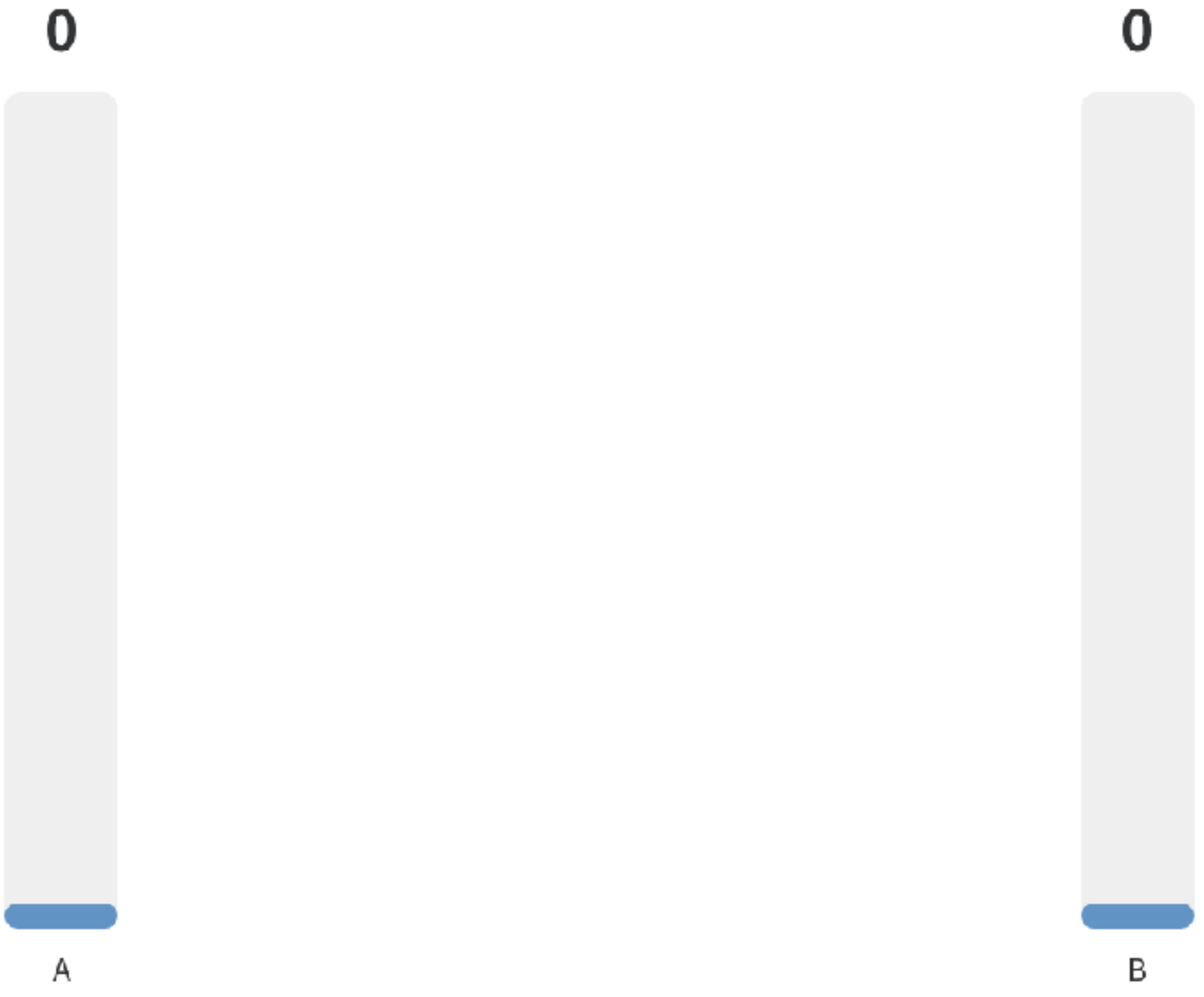
A

```
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];
}
```

0



What if we change the processor?

- If we have an intel processor with a 48KB, 12-way, 64B-blocked L1 cache, which version of code performs better?
  A. Version A, because the code incurs fewer cache misses
  B. Version B, because the code incurs fewer cache misses
  C. Version A, because the code incurs fewer memory references
  D. Version B, because the code incurs fewer memory references
  E. They are about the same

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

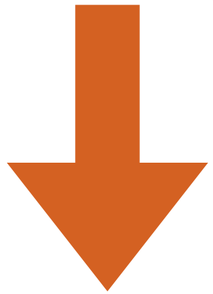
```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
}
```

Loop optimizations

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];
}
```

Loop fission



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];
```

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

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];
}
```

Tiling

Case study: Matrix Multiplication

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

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

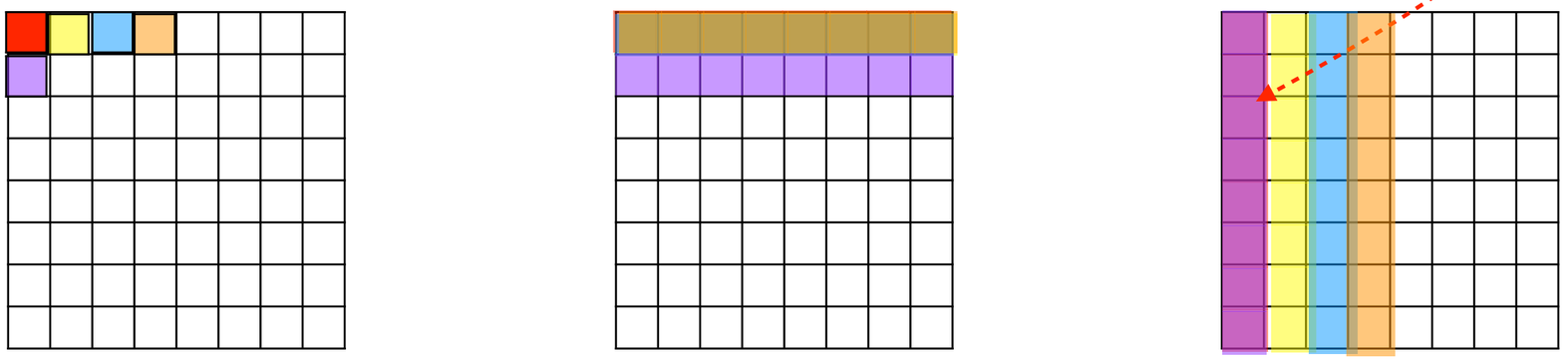
If n=1024, it takes about 1 sec

How long is it take when n=2048?

Matrix Multiplication

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

Very likely a miss if array is large



c a b

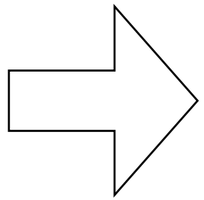
- If each dimension of your matrix is 2048
  - Each row takes 2048*8 bytes = 16KB
  - The L1 $ of intel Core i7 is 32KB, 8-way, 64-byte blocked
  - You can only hold at most 2 rows/columns of each matrix!
  - You need the same row when j increase!

Tiling algorithm for matrix multiplication

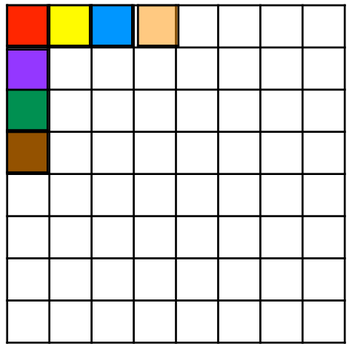
- Discover the cache miss rate
  - valgrind --tool=cachegrind cmd
    - cachegrind is a tool profiling the cache performance
  - Performance counter
    - Intel® Performance Counter Monitor http://www.intel.com/software/pcm/

Tiling algorithm for matrix multiplication

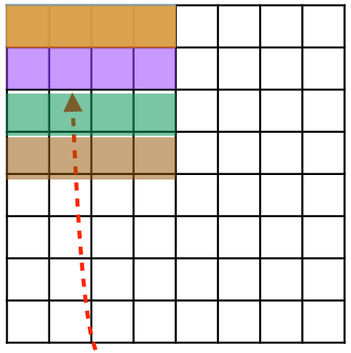
```
for(i = 0; i < ARRAY_SIZE; i++) {
  for(j = 0; j < ARRAY_SIZE; j++) {
    for(k = 0; k < ARRAY_SIZE; k++) {
      c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```



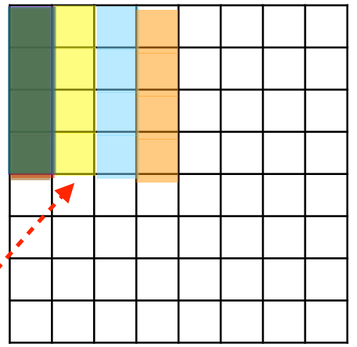
```
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
  for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
    for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
        for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
          for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
            for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
              c[ii][jj] += a[ii][kk]*b[kk][jj];
    }
  }
}
```



c



a



b

You only need to hold these sub-matrices in your cache

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

What kind(s) of misses can tiling algorithm remove?

- Comparing the naive algorithm and tiling algorithm on matrix multiplication, what kind of misses does tiling algorithm help to remove? (assuming an intel Core i7)

Naive

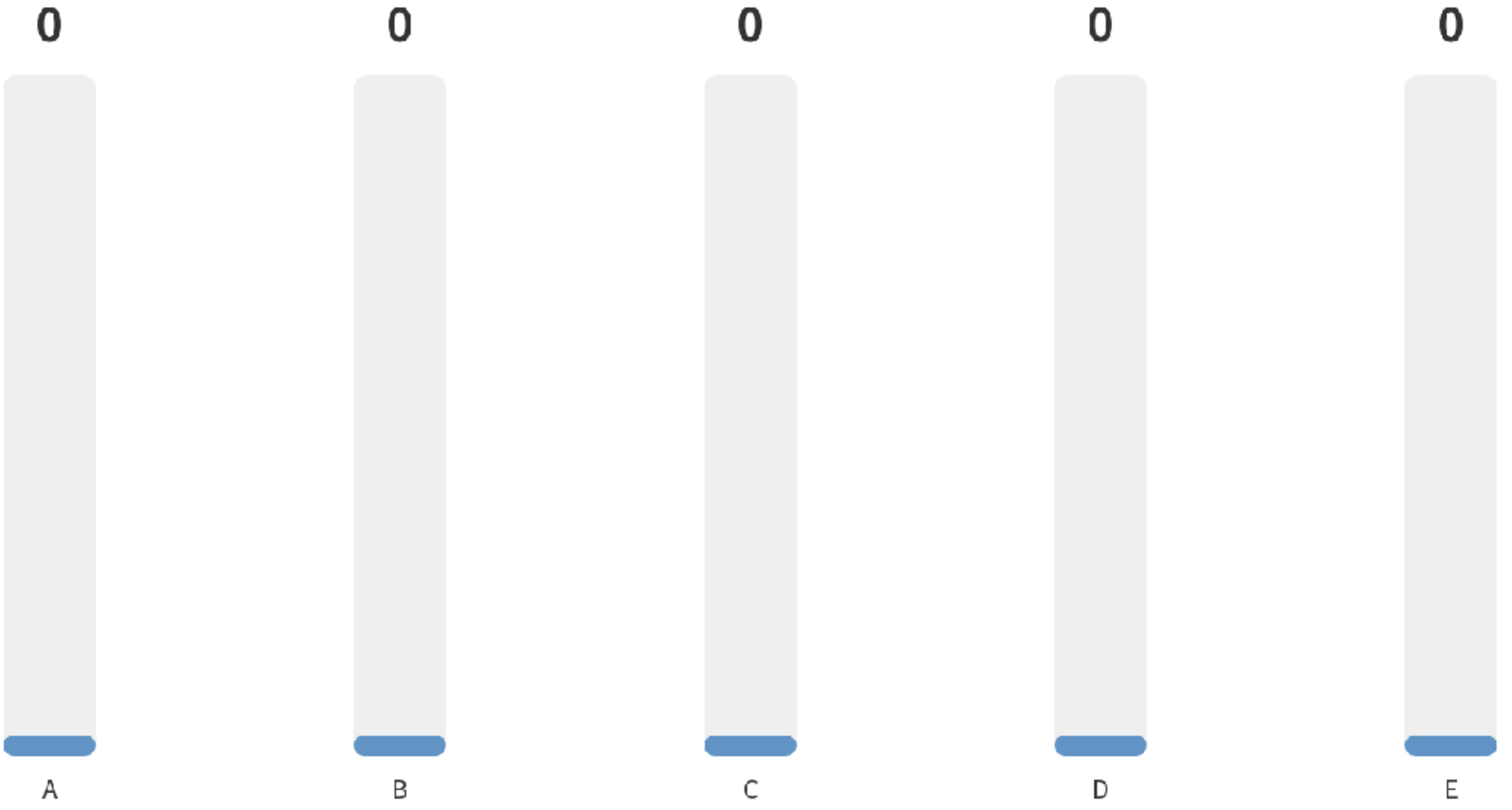
```
for(i = 0; i < ARRAY_SIZE; i++) {
  for(j = 0; j < ARRAY_SIZE; j++) {
    for(k = 0; k < ARRAY_SIZE; k++) {
      c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

Block

```
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
  for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
    for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
      for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
        for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
          for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
            c[ii][jj] += a[ii][kk]*b[kk][jj];
    }
  }
}
```

A. Compulsory miss
B. Capacity miss
C. Conflict miss
D. Capacity & conflict miss
E. Compulsory & conflict miss

0



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

What kind(s) of misses can tiling algorithm remove?

- Comparing the naive algorithm and tiling algorithm on matrix multiplication, what kind of misses does tiling algorithm help to remove? (assuming an intel Core i7)

Naive

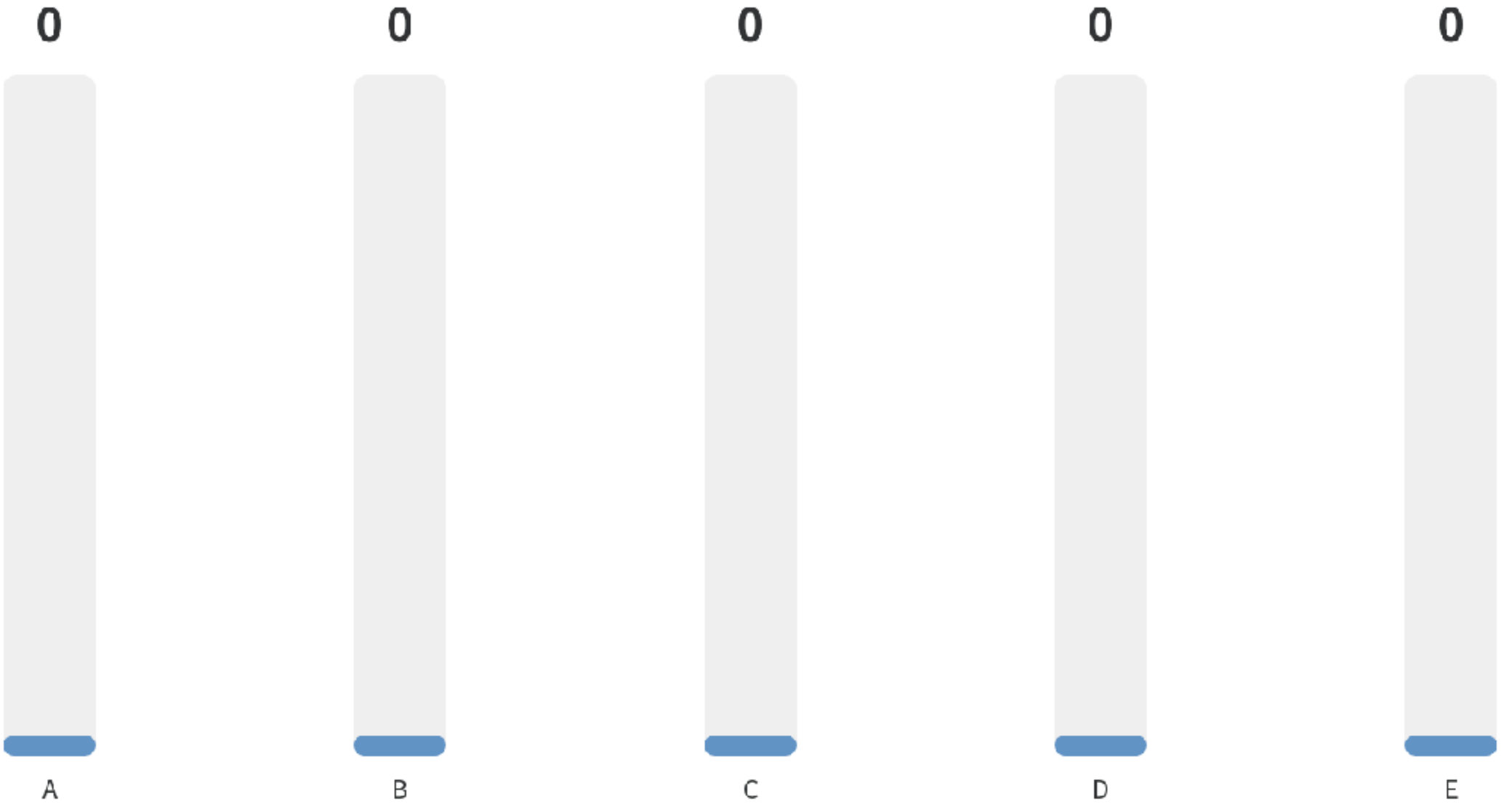
```
for(i = 0; i < ARRAY_SIZE; i++) {
  for(j = 0; j < ARRAY_SIZE; j++) {
    for(k = 0; k < ARRAY_SIZE; k++) {
      c[i][j] += a[i][k]*b[k][j];
    }
  }
}
```

Block

```
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
  for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
    for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
      for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
        for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
          for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
            c[ii][jj] += a[ii][kk]*b[kk][jj];
    }
  }
}
```

A. Compulsory miss
B. Capacity miss
C. Conflict miss
D. Capacity & conflict miss
E. Compulsory & conflict miss

0



What kind(s) of misses can tiling algorithm remove?

- Comparing the naive algorithm and tiling algorithm on matrix multiplication, what kind of misses does tiling algorithm help to remove? (assuming an intel Core i7)

Naive

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

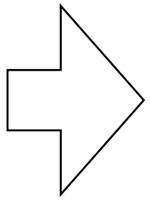
Block

```
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
  for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
    for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
      for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
        for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
          for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
            c[ii][jj] += a[ii][kk]*b[kk][jj];
    }
  }
}
```

A. Compulsory miss
B. Capacity miss
C. Conflict miss
D. Capacity & conflict miss
E. Compulsory & conflict miss

Matrix Transpose

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

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

What kind(s) of misses can matrix transpose remove?

- By transposing a matrix, the performance of matrix multiplication can be further improved. What kind(s) of cache misses does matrix transpose help to remove?

Block

```
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
  for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
    for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
      for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
        for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
          for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
            c[ii][jj] += a[ii][kk]*b[kk][jj];
    }
  }
}
```

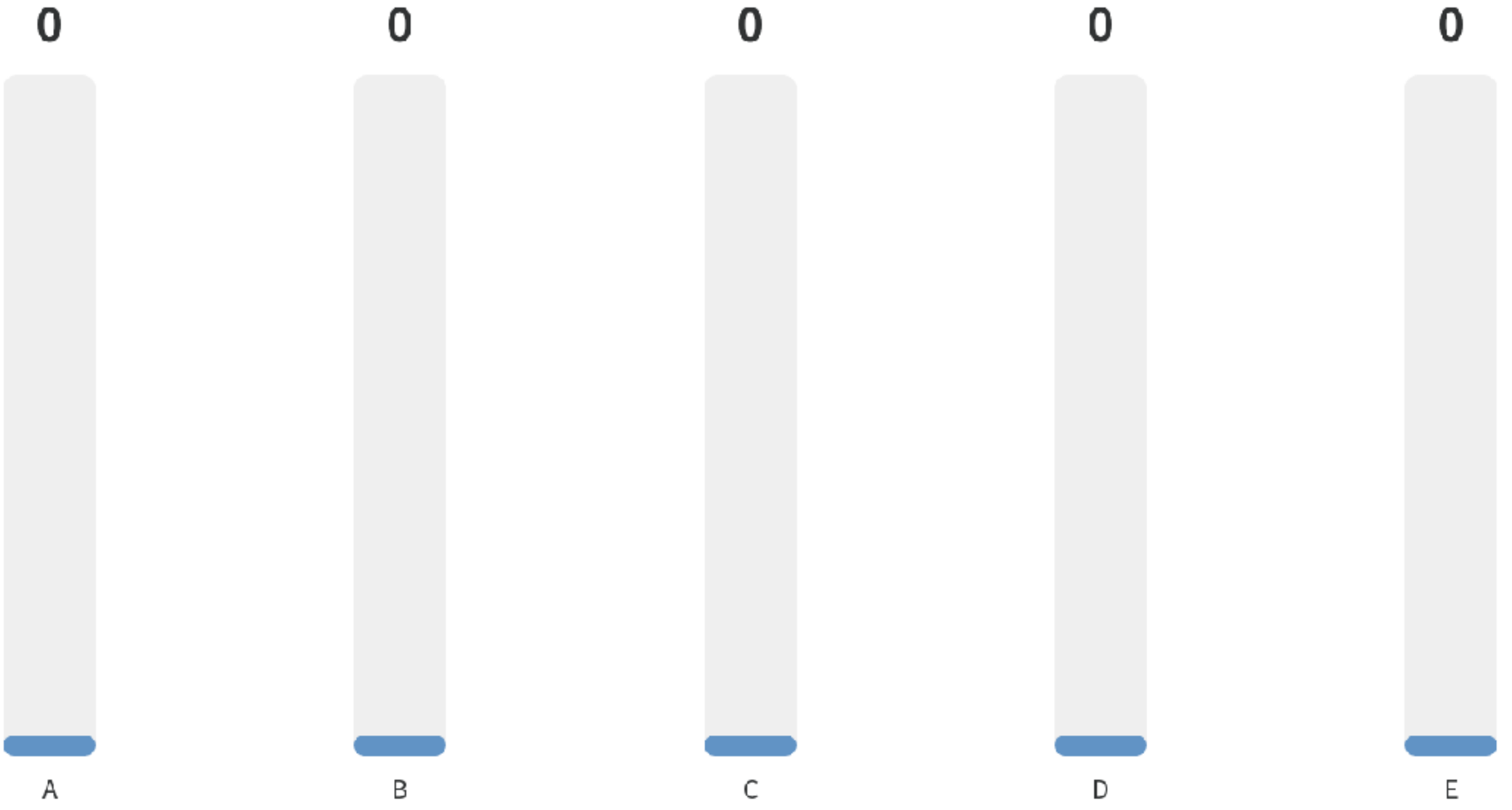
A. Compulsory miss
B. Capacity miss
C. Conflict miss
D. Capacity & conflict miss
E. Compulsory & conflict miss

Block + Transpose

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

0



What kind(s) of misses can matrix transpose remove?

- By transposing a matrix, the performance of matrix multiplication can be further improved. What kind(s) of cache misses does matrix transpose help to remove?

Block

```
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
  for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
    for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
      for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
        for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
          for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
            c[ii][jj] += a[ii][kk]*b[kk][jj];
    }
  }
}
```

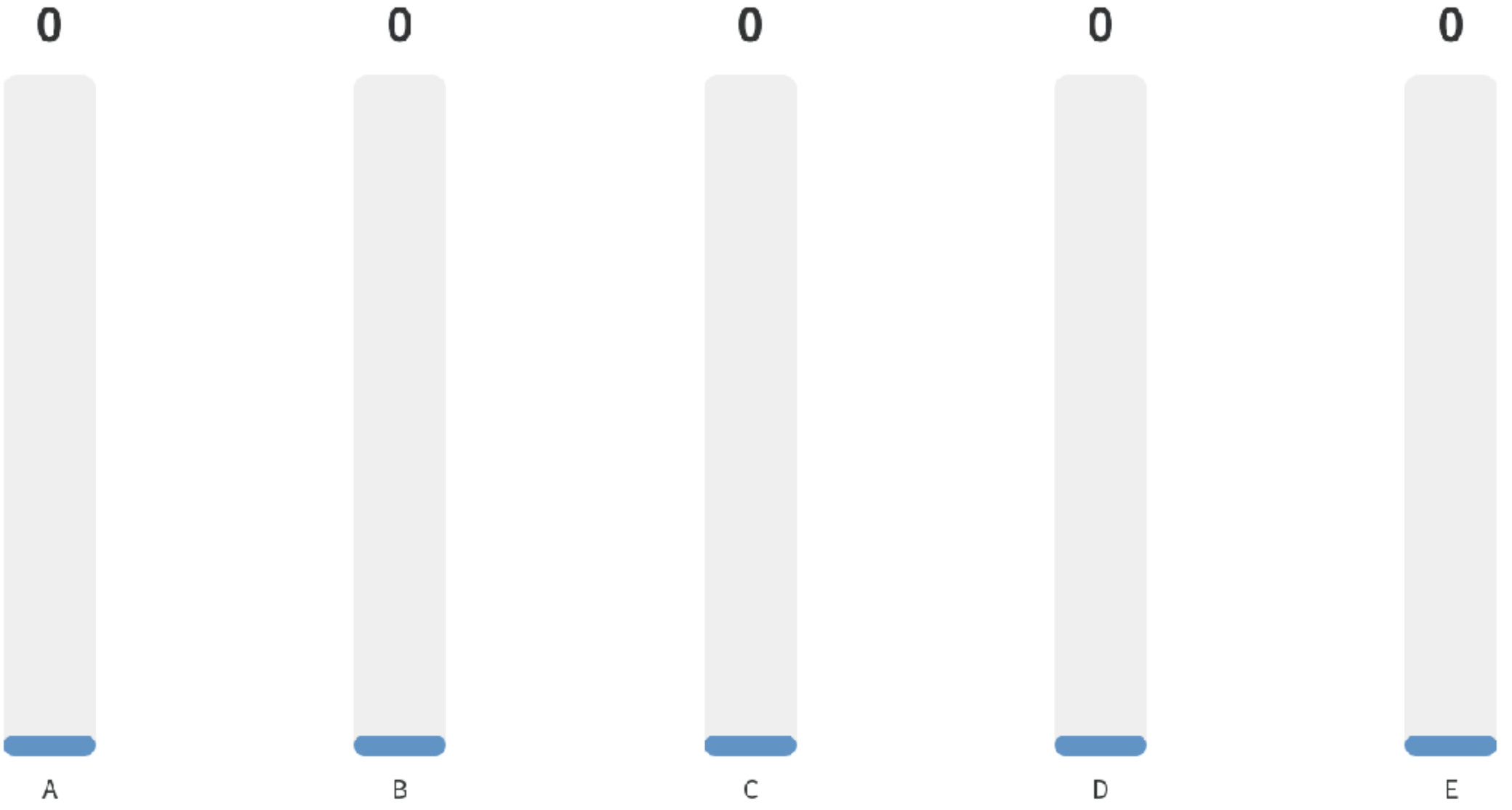
Block + Transpose

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

A. Compulsory miss
B. Capacity miss
C. Conflict miss
D. Capacity & conflict miss
E. Compulsory & conflict miss

0



What kind(s) of misses can matrix transpose remove?

- By transposing a matrix, the performance of matrix multiplication can be further improved. What kind(s) of cache misses does matrix transpose help to remove?

Block

```
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {
  for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {
    for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {
      for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)
        for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)
          for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)
            c[ii][jj] += a[ii][kk]*b[kk][jj];
    }
  }
}
```

A. Compulsory miss
B. Capacity miss
C. Conflict miss
D. Capacity & conflict miss
E. Compulsory & conflict miss

Block + Transpose

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

Summary of Software Optimizations

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

Basic Hardware Optimization in Improving 3Cs

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 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

3C & Misses

0

A
0%

B
0%

C
0%

D
0%

E
0%

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 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

3C & Misses — group

0

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

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 < 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]
}
```

Small Buffers

0

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 < 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]
}
```

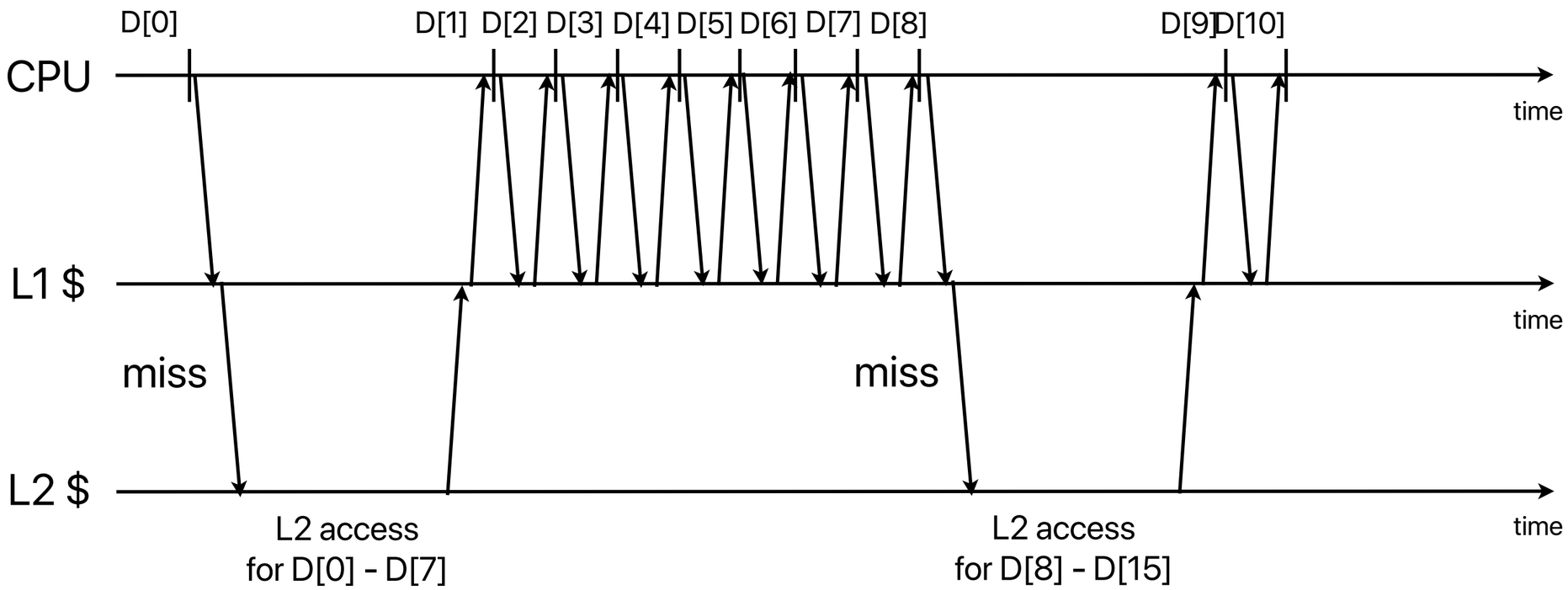
Small Buffers — group

0

Prefetching

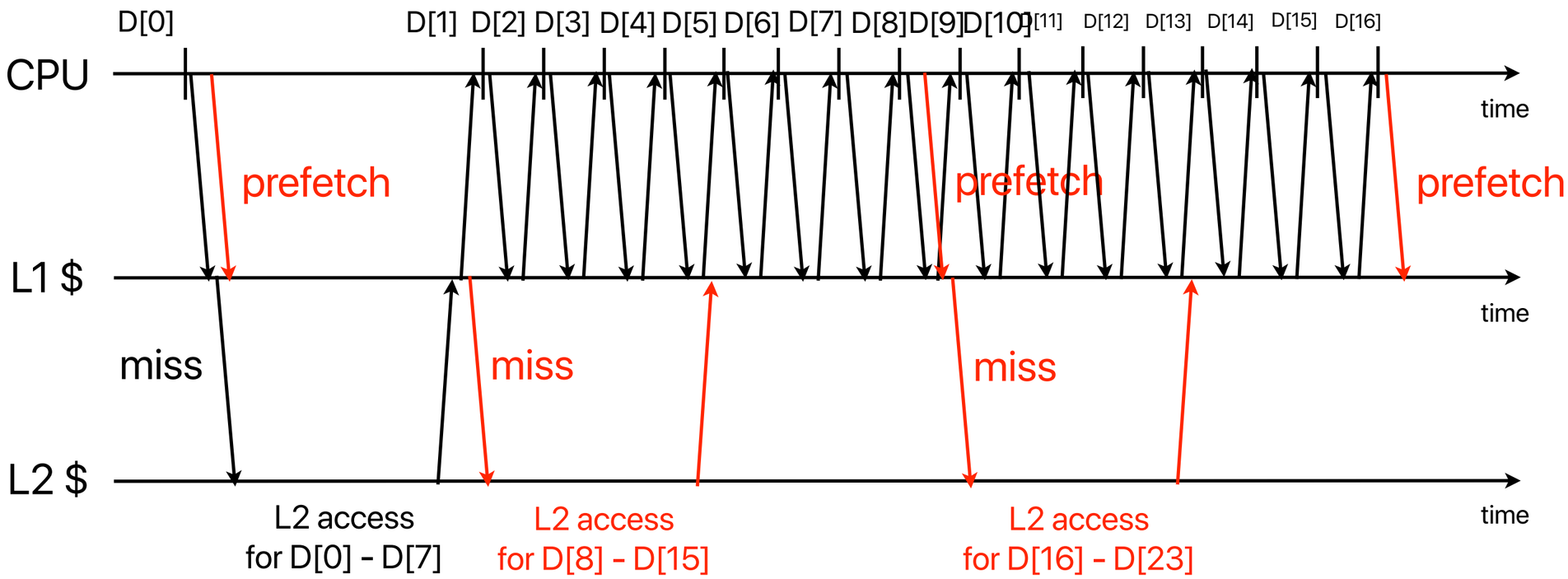
Characteristic of memory accesses

```
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 miss_data_address+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

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

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

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();
```

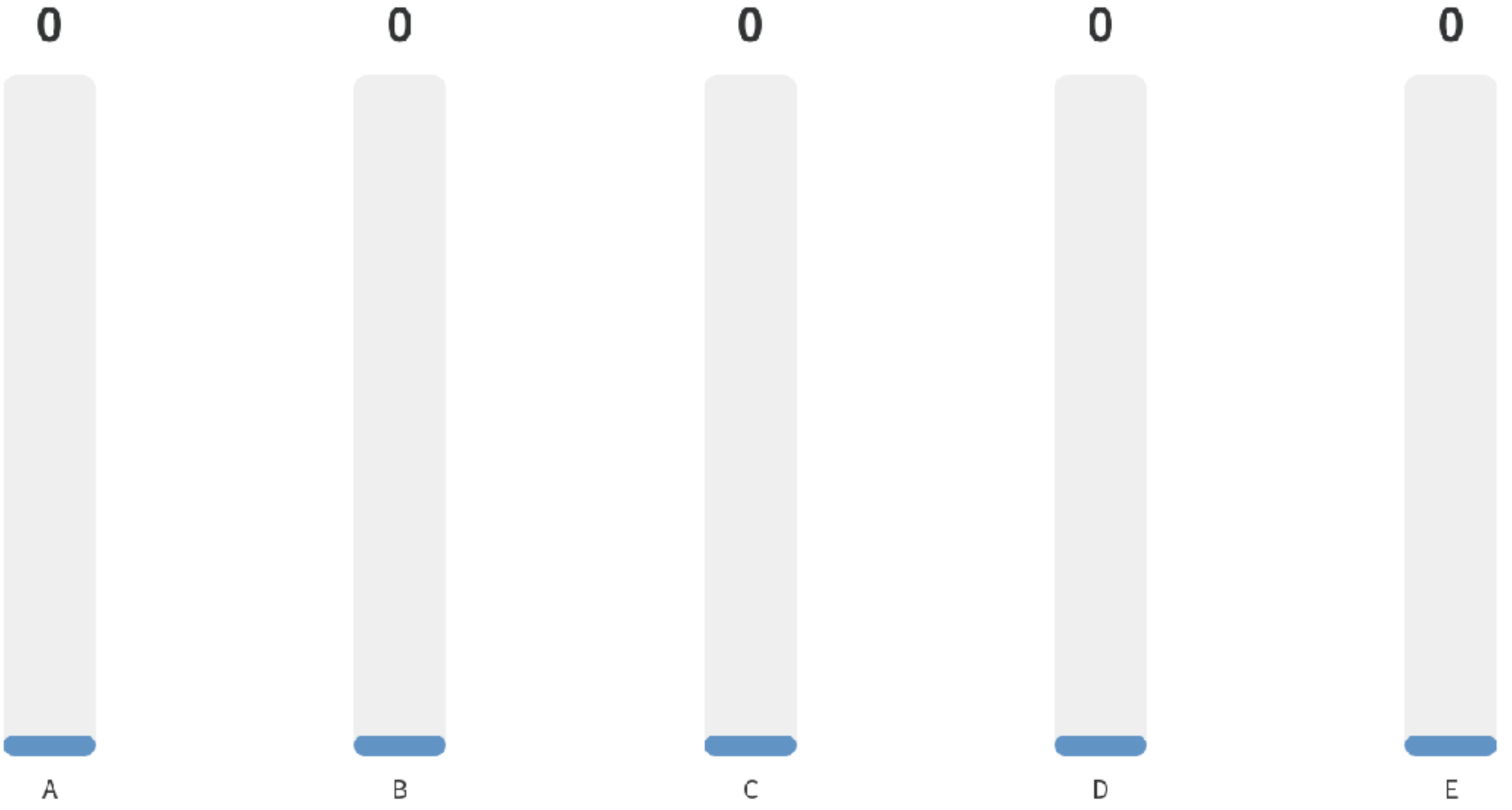
(3)
```
while (root != NULL){
        if (key > root->data)
            root = root->right;

        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

0



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

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();
```

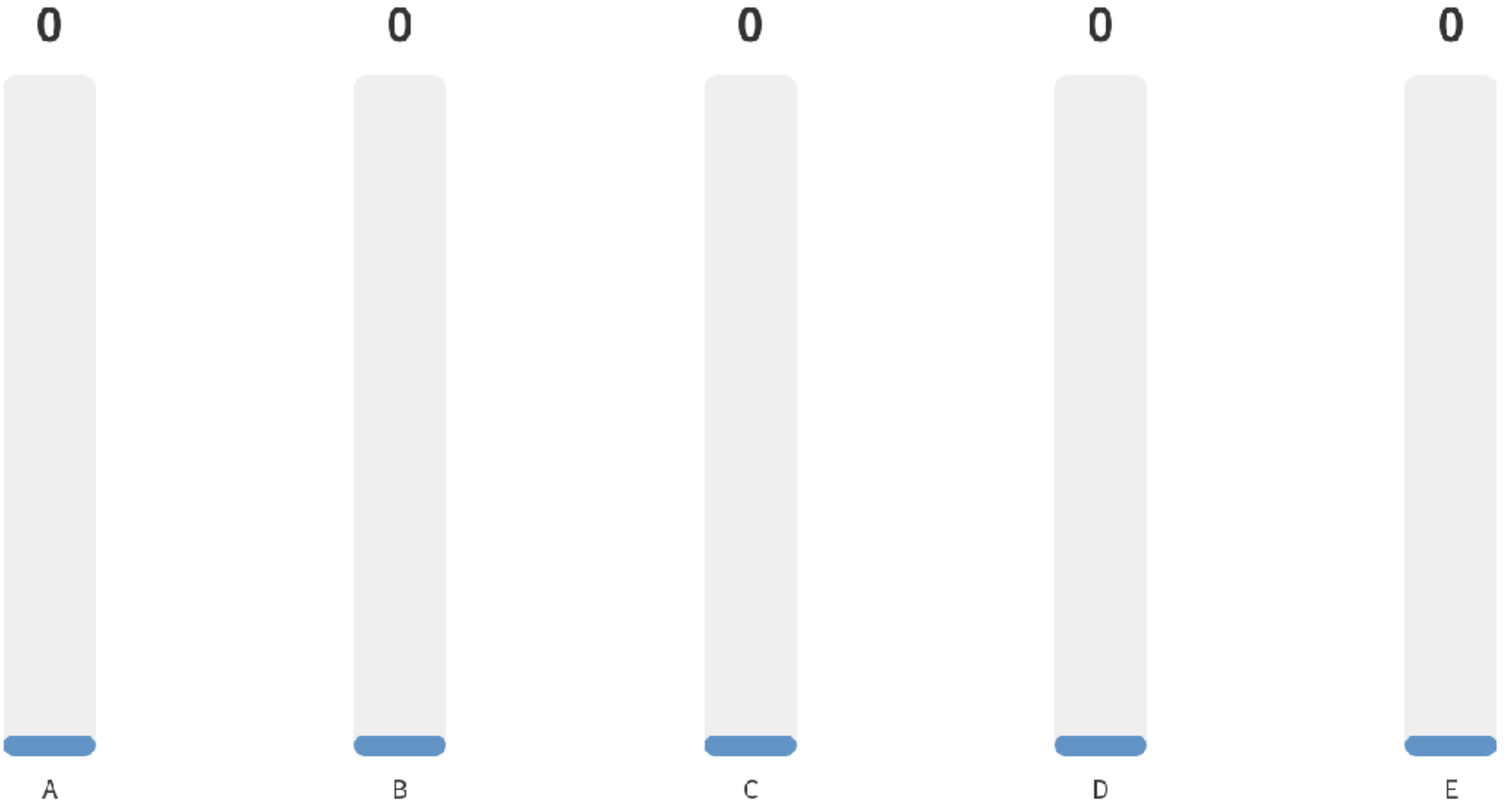
(3)
```
while (root != NULL){
        if (key > root->data)
            root = root->right;

        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

0



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();
```

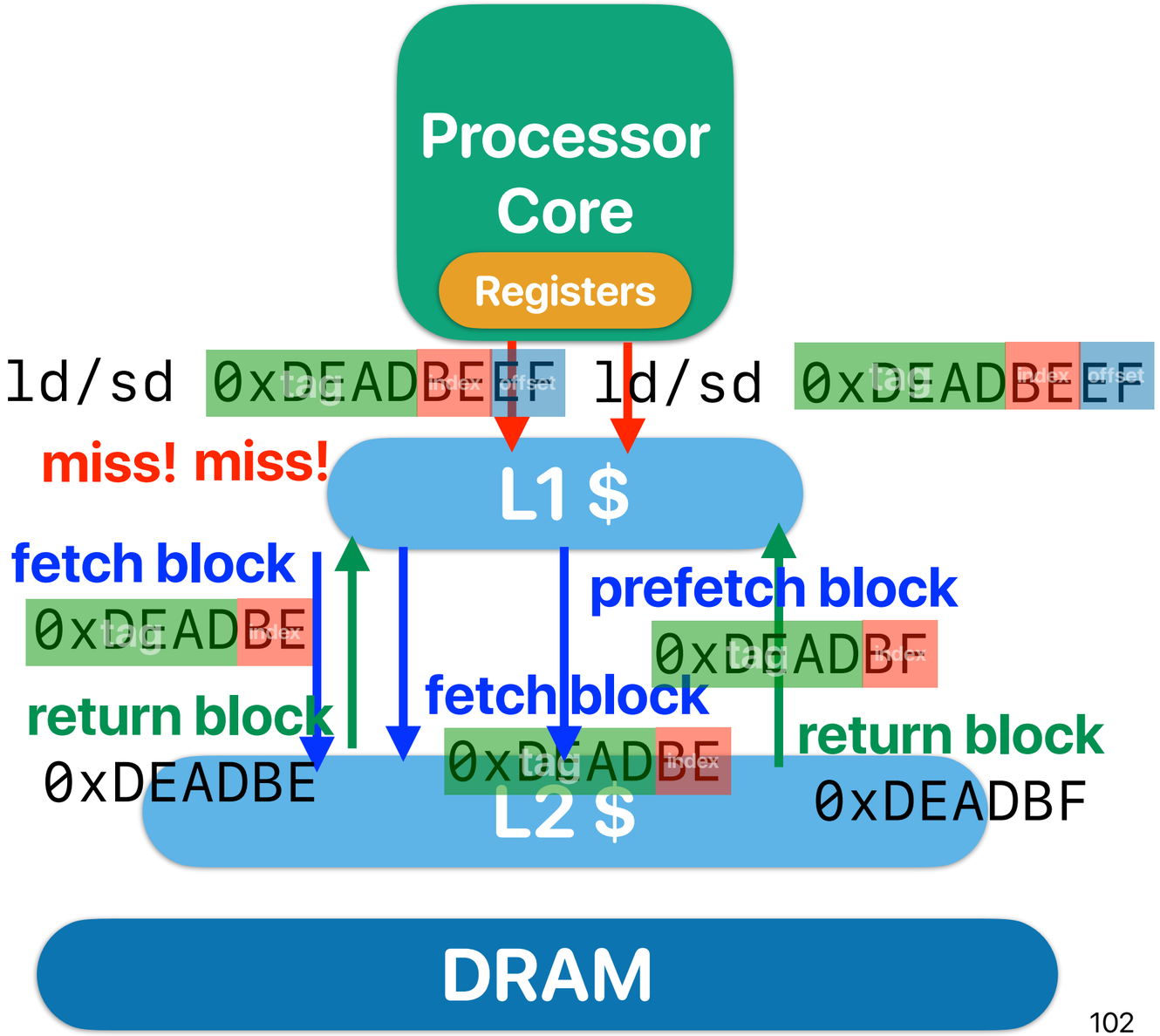
(3)
```
while (root != NULL){
        if (key > root->data)
            root = root->right;

        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

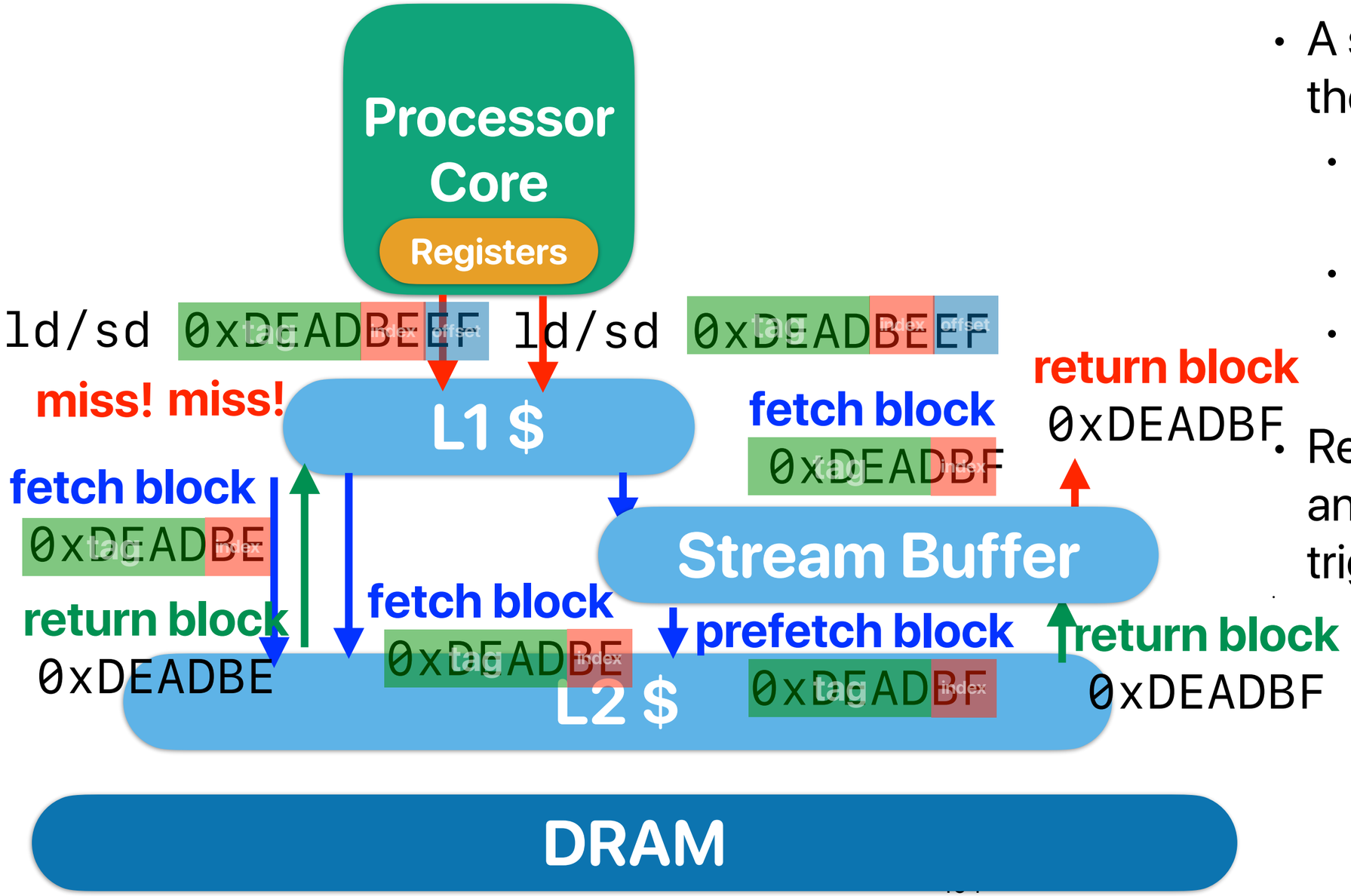
tag index offset

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?	Prefetch
a[0]	0x10000	0b00010000000000000000	0x8	0x0	Miss		a[8–15]
b[0]	0x20000	0b00100000000000000000	0x10	0x0	Miss		b[8–15]
c[0]	0x30000	0b00110000000000000000	0x18	0x0	Miss		c[8–15]
d[0]	0x40000	0b01000000000000000000	0x20	0x0	Miss		d[8–15]
e[0]	0x50000	0b01010000000000000000	0x28	0x0	Miss	a[0–7]	e[8–15]
a[1]	0x10008	0b00010000000000001000	0x8	0x0	Miss	b[0–7]	
b[1]	0x20008	0b00100000000000001000	0x10	0x0	Miss	c[0–7]	
c[1]	0x30008	0b00110000000000001000	0x18	0x0	Miss	d[0–7]	
d[1]	0x40008	0b01000000000000001000	0x20	0x0	Miss	e[0–7]	
e[1]	0x50008	0b01010000000000001000	0x28	0x0	Miss	a[0–7]	
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

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

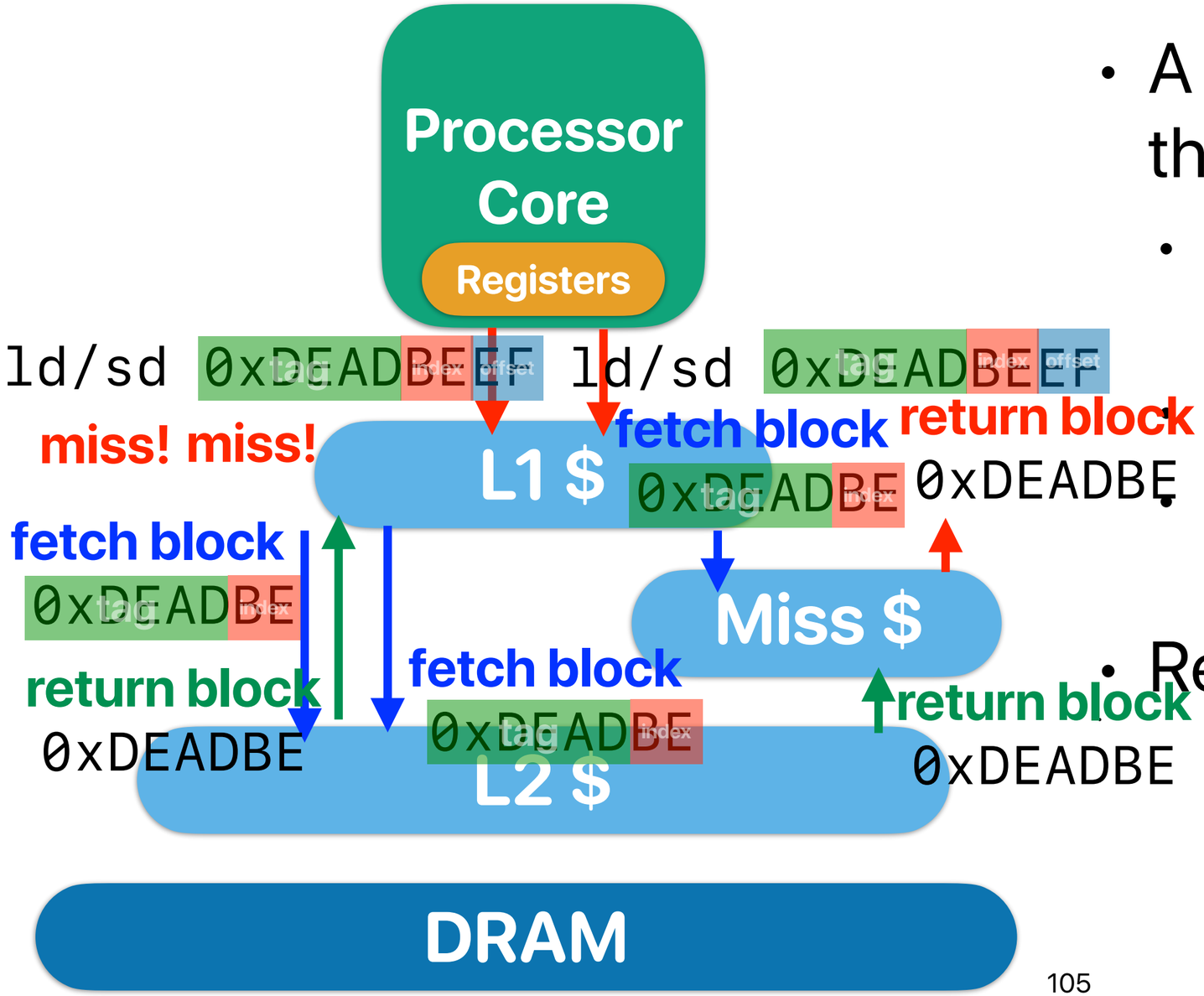
100% miss rate!

Stream buffer



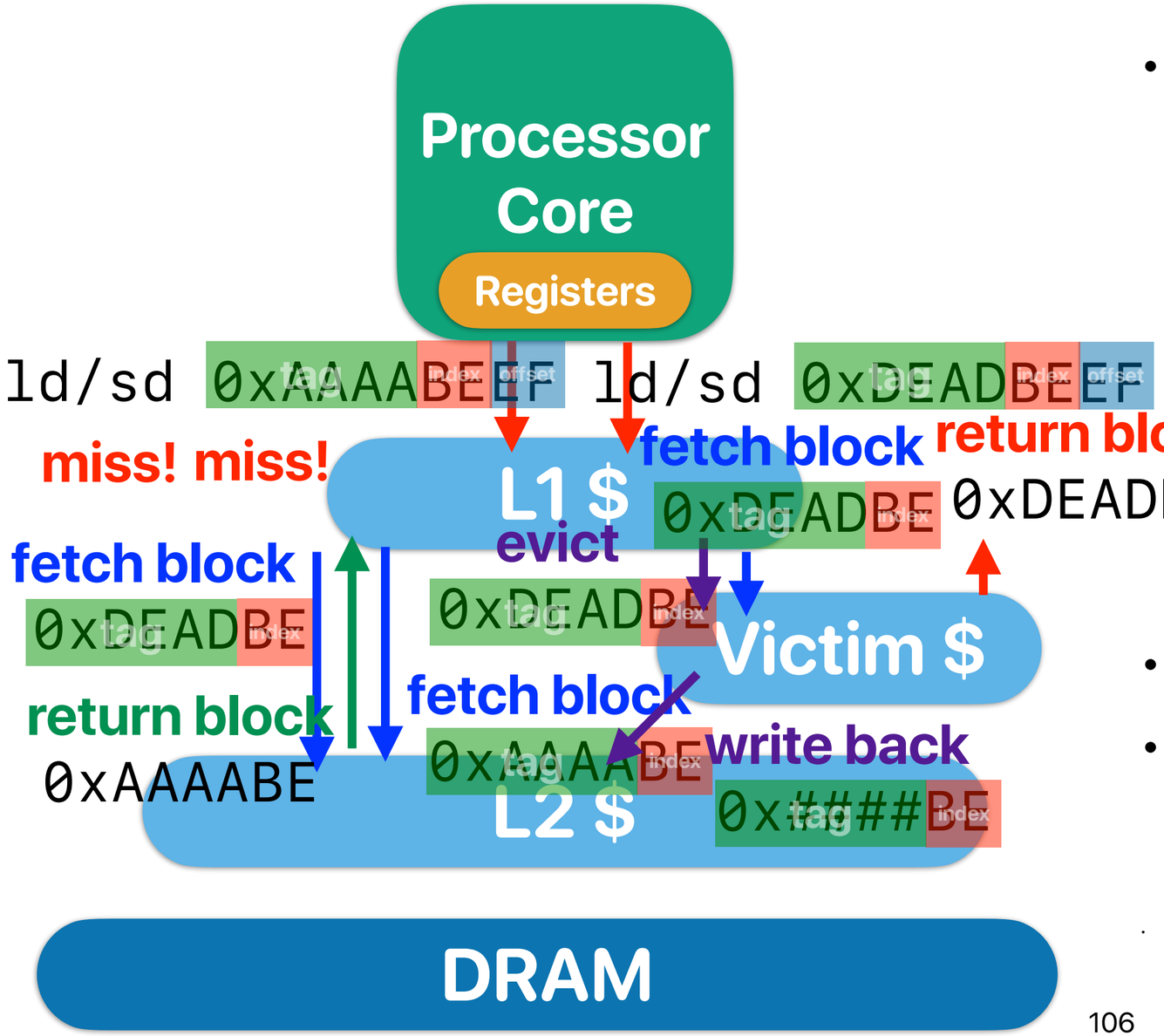
- A small cache that captures the prefetched blocks
  - Can be built as fully associative since it's small
  - Consult when there is a miss
  - Retrieve the block if found in the stream buffer
- Reduce compulsory misses and avoid conflict misses triggered by prefetching

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

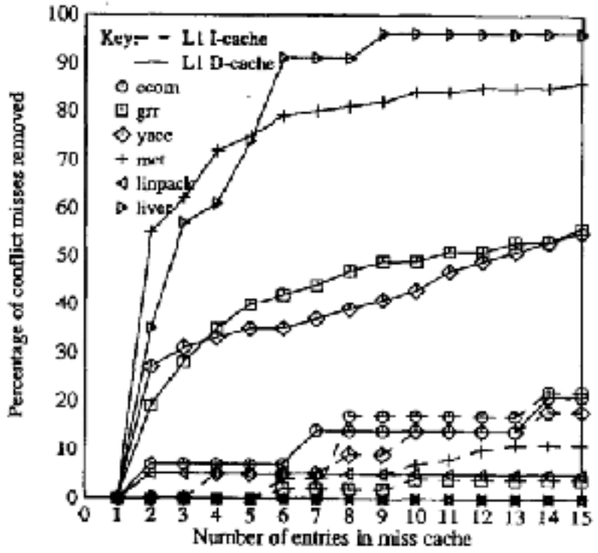


Figure 3-3: Conflict misses removed by miss caching

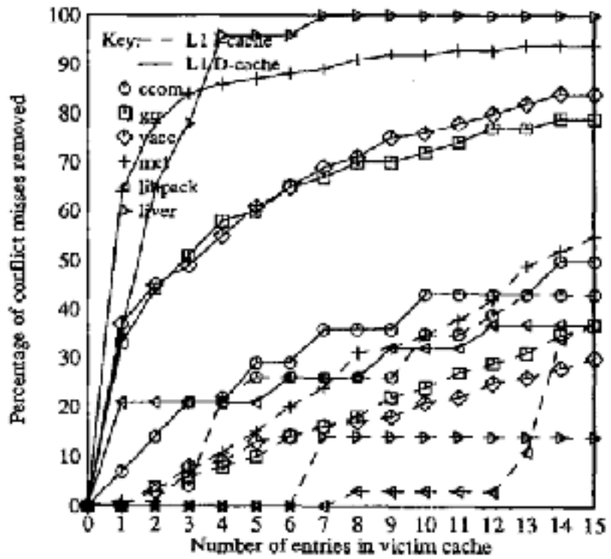


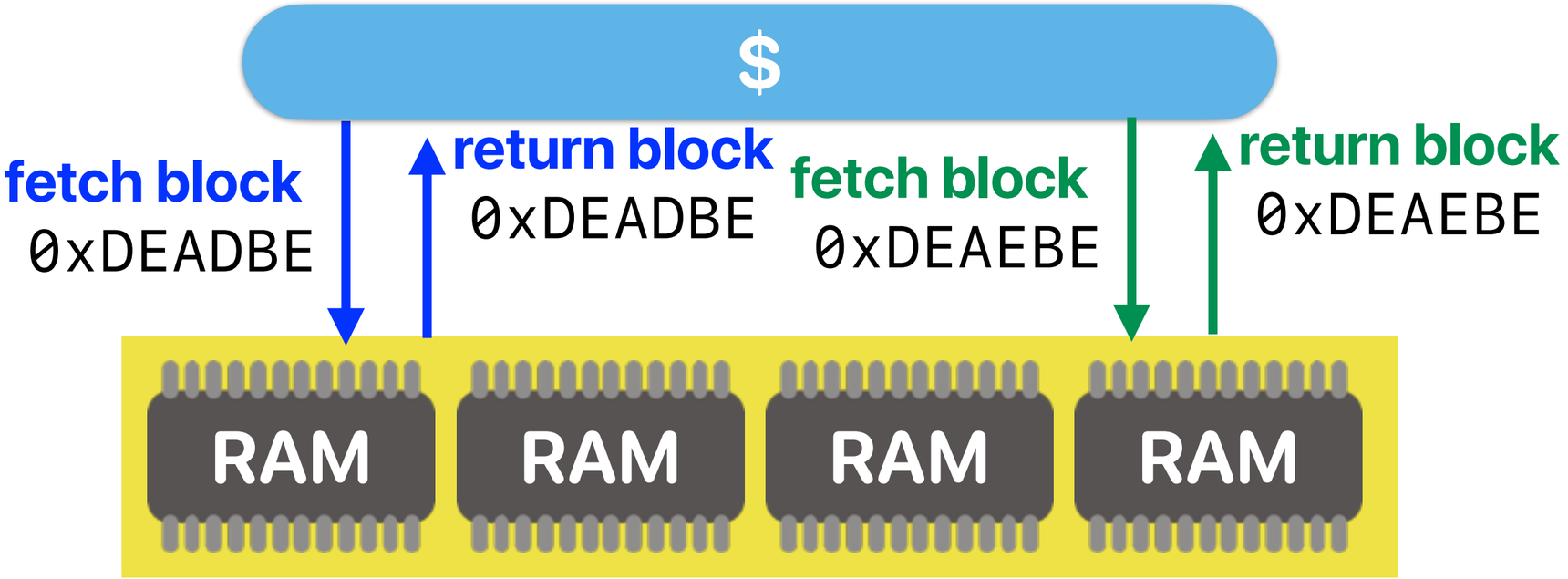
Figure 3-5: Conflict misses removed by victim caching

Which of the following schemes can help Tegra?

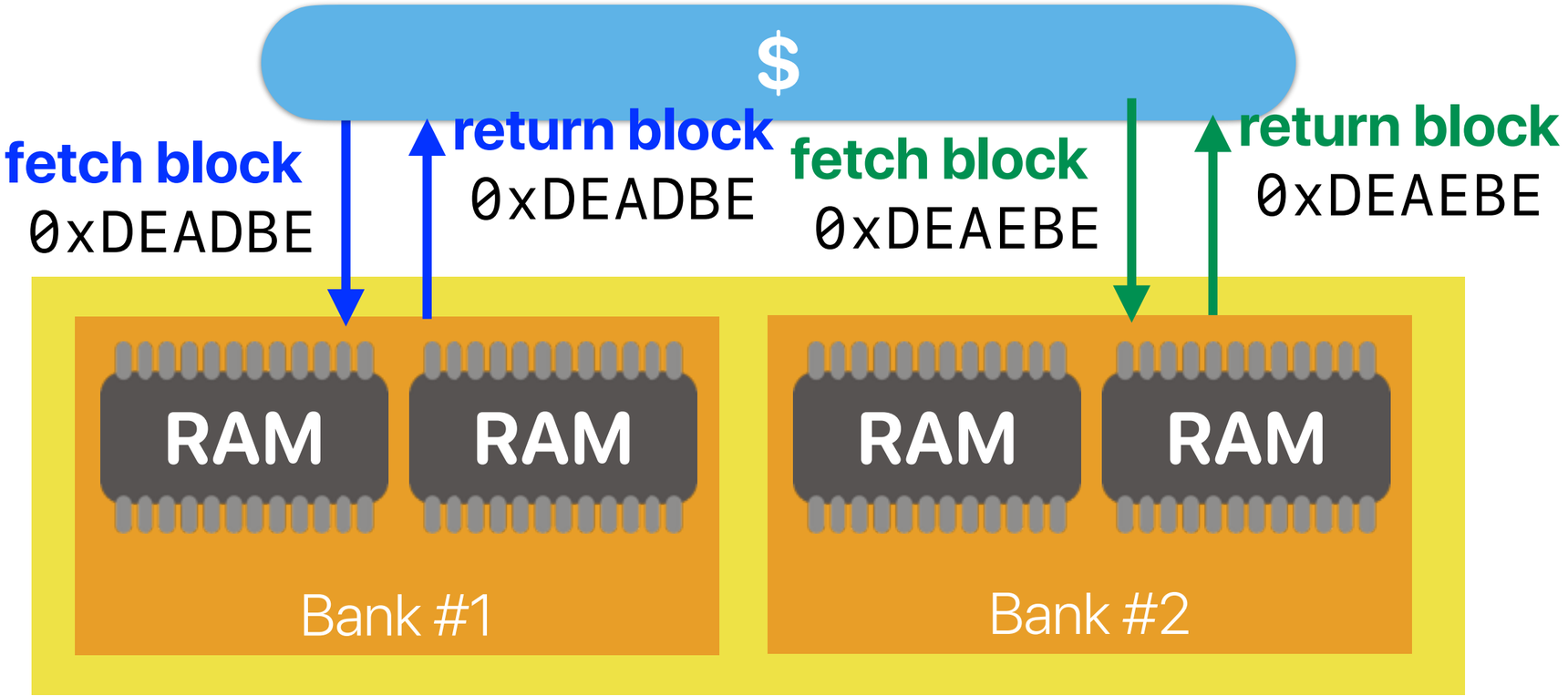
- How many of the following schemes mentioned in "improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers" would help NVIDIA's Tegra for the code in the previous slide?
  - ① Missing cache — help improving conflict misses
  - ② Victim cache — 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

Advanced Hardware Techniques in Improving Memory Performance

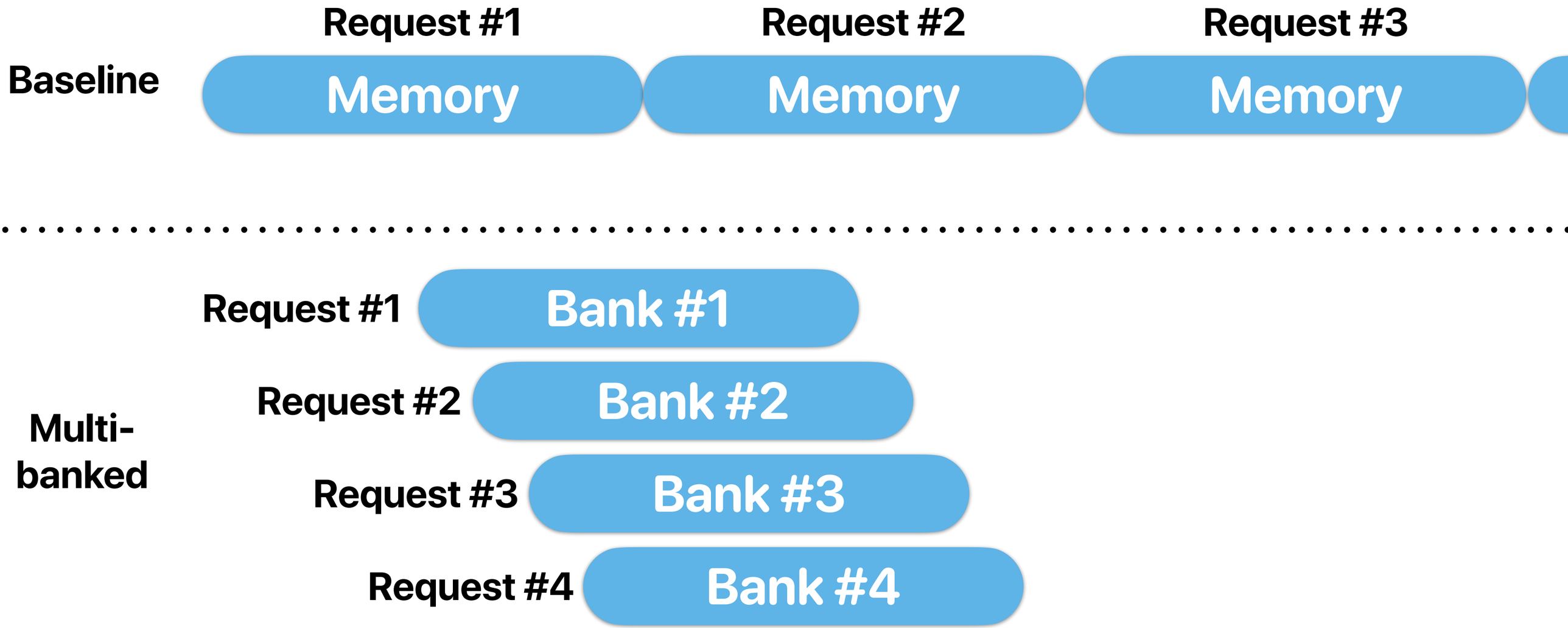
Blocking cache



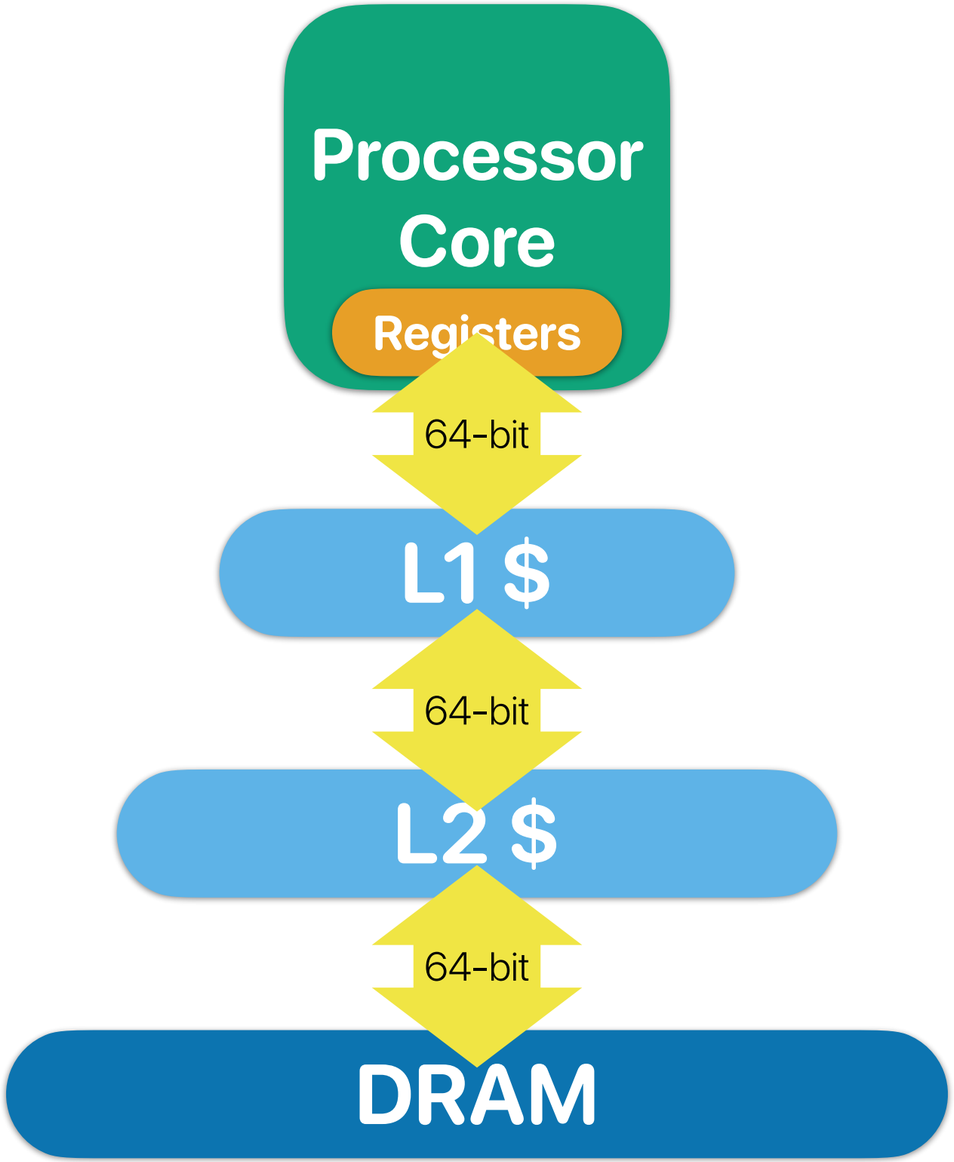
Multibanks & non-blocking caches



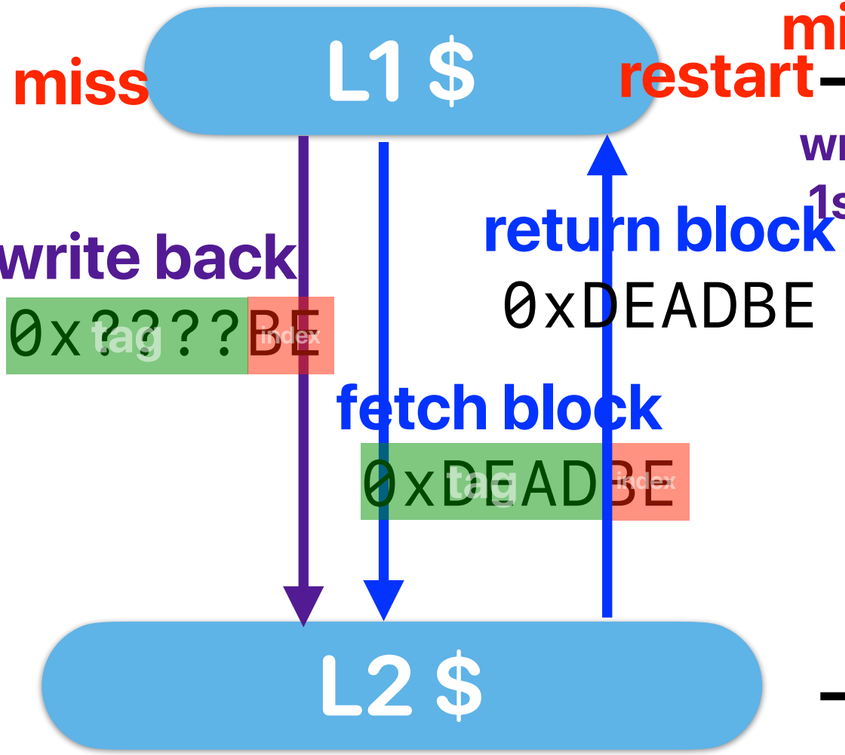
Pipelined access and multi-banked caches

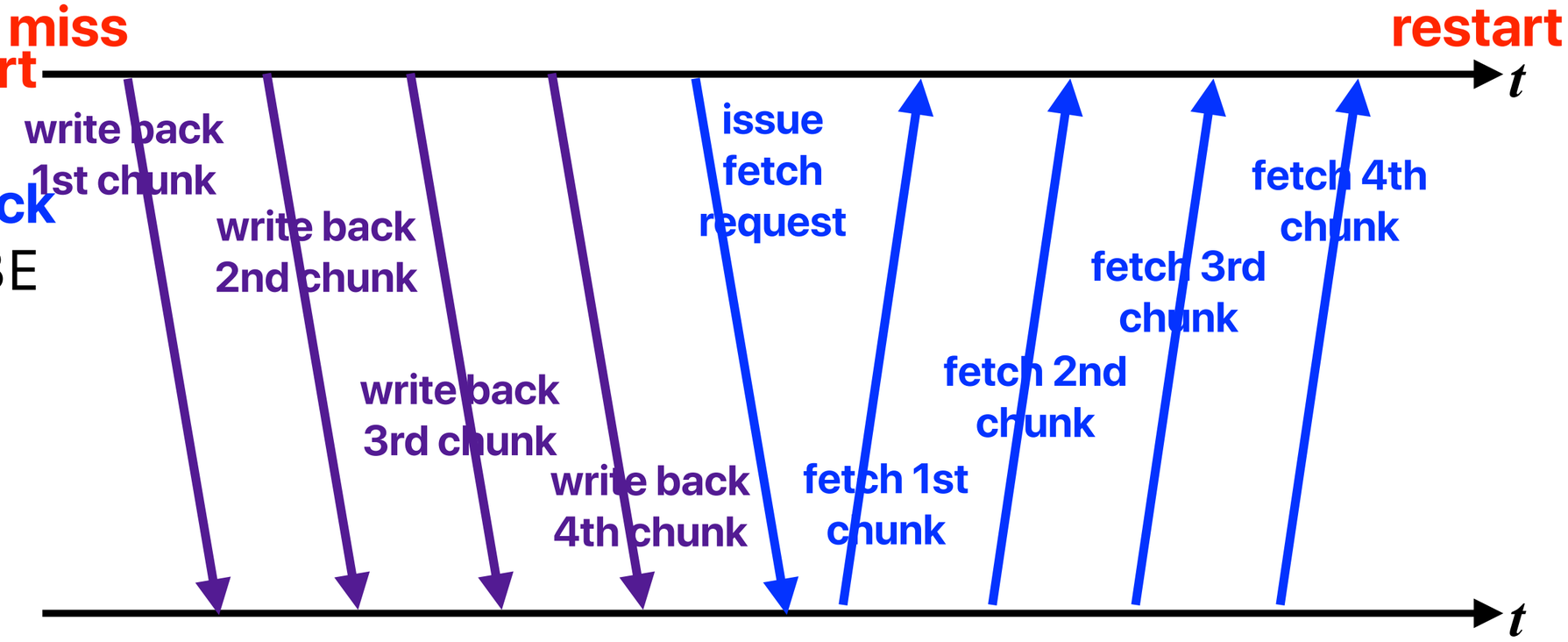


The bandwidth between units is limited



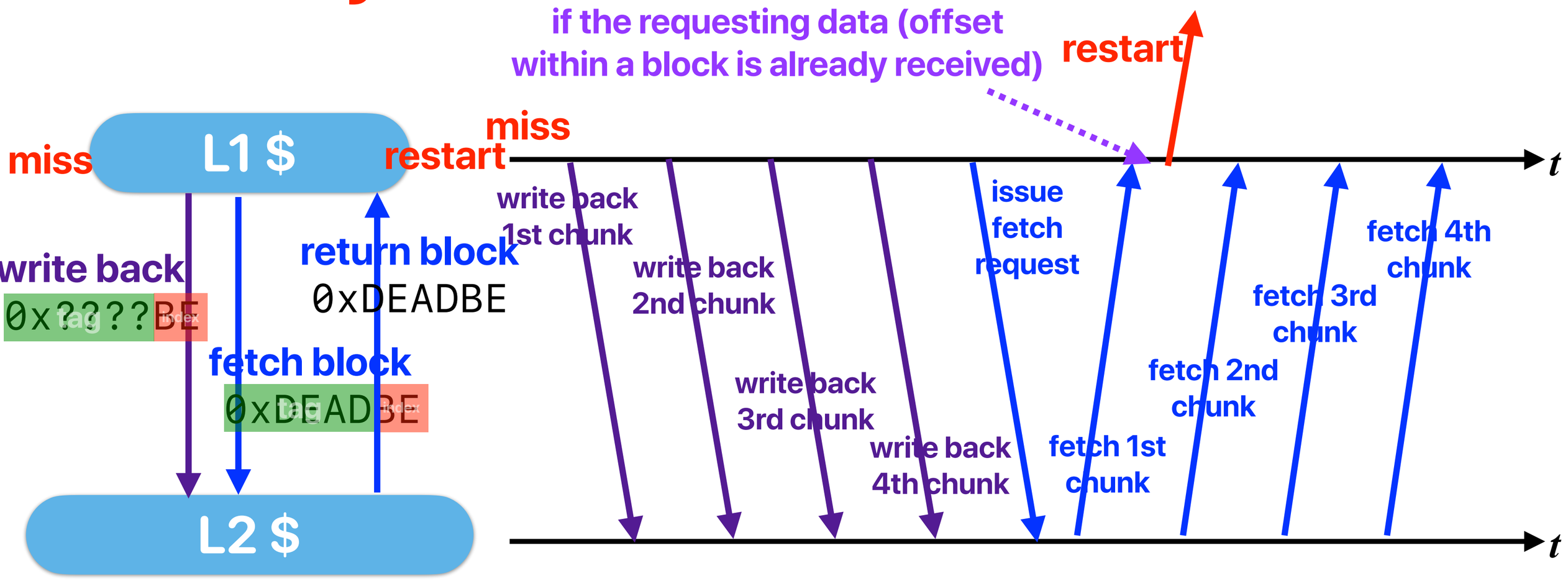
When we handle a miss





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

Early Restart and Critical Word First

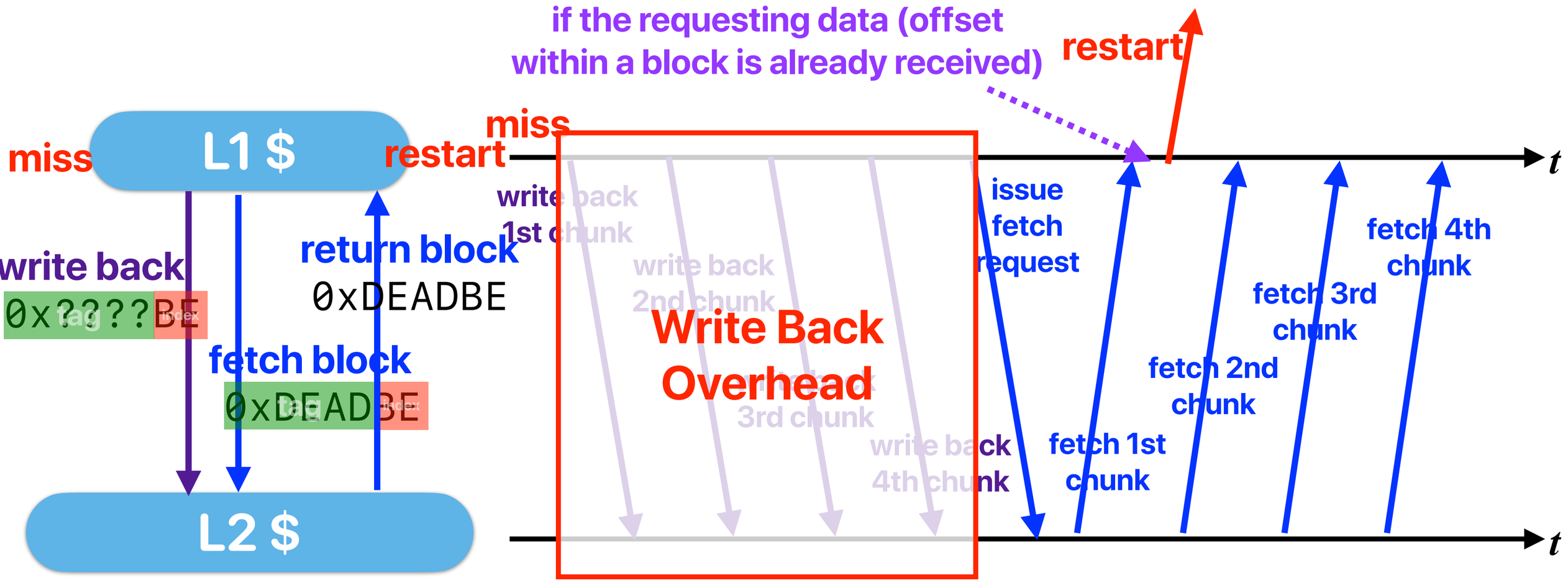


assume the bus between L1/L2 only allows a quarter of the cache block go through it

Early Restart and Critical Word First

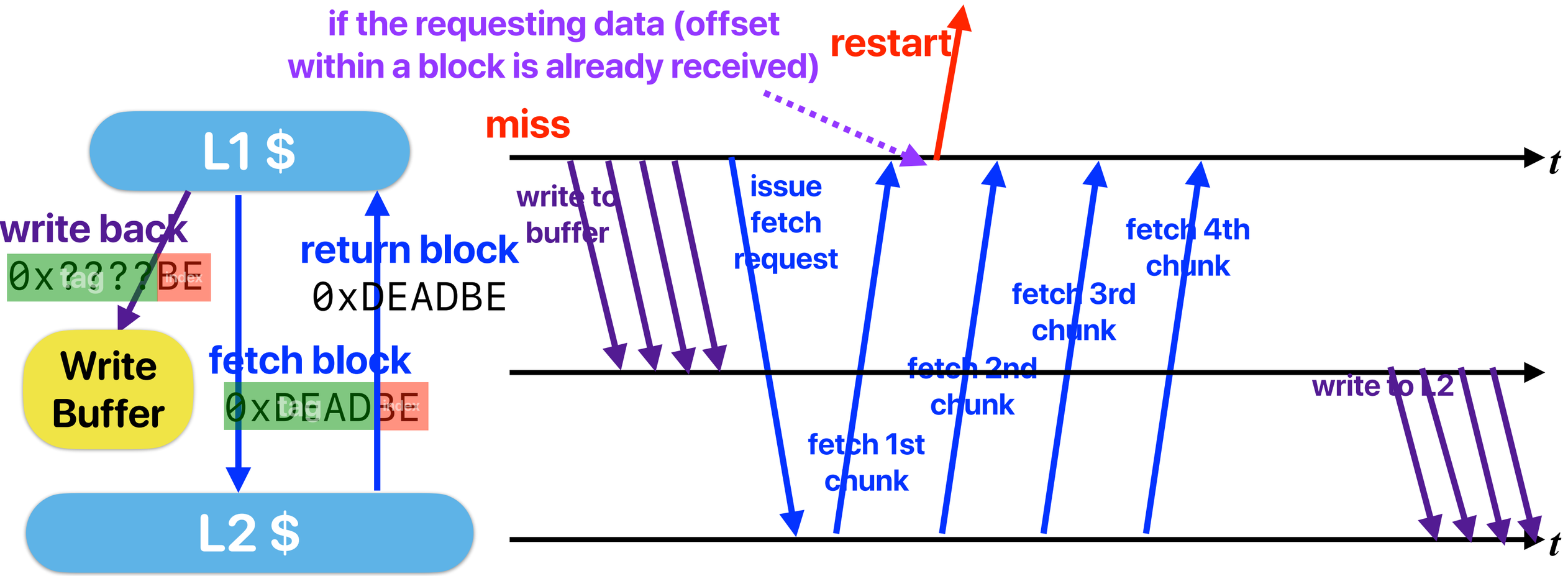
- Don’t wait for full block to be loaded before restarting CPU
  - Early restart—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
  - Critical Word First—Request the missed word first from memory and send it to the CPU as soon as it arrives; let 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.

Summary of Optimizations

- Regarding the following cache optimizations, how many of them would help improve miss rate?
  - ① Non-blocking/pipelined/multibanked cache
  - ② Critical word first and early restart
  - ③ Prefetching
  - ④ Write buffer

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

Hardware Optimizations

0

Summary of Optimizations

- Regarding the following cache optimizations, how many of them would help improve miss rate?
  - ① Non-blocking/pipelined/multibanked cache
  - ② Critical word first and early restart
  - ③ Prefetching
  - ④ Write buffer

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

Hardware Optimizations — group

0

Summary of Optimizations

- Regarding the following cache optimizations, how many of them would help improve miss rate?
  - ① Non-blocking/pipelined/multibanked cache **Miss penalty/Bandwidth**
  - ② Critical word first and early restart **Miss penalty**
  - ③ Prefetching **Miss rate (compulsory)**
  - ④ Write buffer **Miss penalty**

  A. 0

  B. 1

  C. 2

  D. 3

  E. 4

Summary of Optimizations

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


Q & A

Announcement

- Reading quiz #4 due **next next Thursday** before the lecture
- Check your participation grade on https://www.escalab.org/my_grades/ (You may also find the link of the course website)
- Assignment #3 is up. Due in next Thursday (in 7 days)
- Programming assignments are perfectly linked to lectures
  - The solution of programming assignment #2 can be found in the first and the second lecture
  - You should be inspired today for PA #3
- Plagiarism:
  - You cannot directly use any code that you found online due to copyright issues
    - You will be sued if you use code from others to in-production software
    - You need to practice how to avoid those issues as a student
  - Please review the course website and the slide from the first lecture
    - You have to give "credits" to who you have consulted — but it does not mean you can use their code
    - Please review ACM's policy on the use of generative AI
      - https://www.acm.org/publications/policies/frequently-asked-questions
      - https://www.acm.org/binaries/content/assets/public-policy/ustpc-approved-generative-ai-principles

Computer
Science &
Engineering

203

つづく