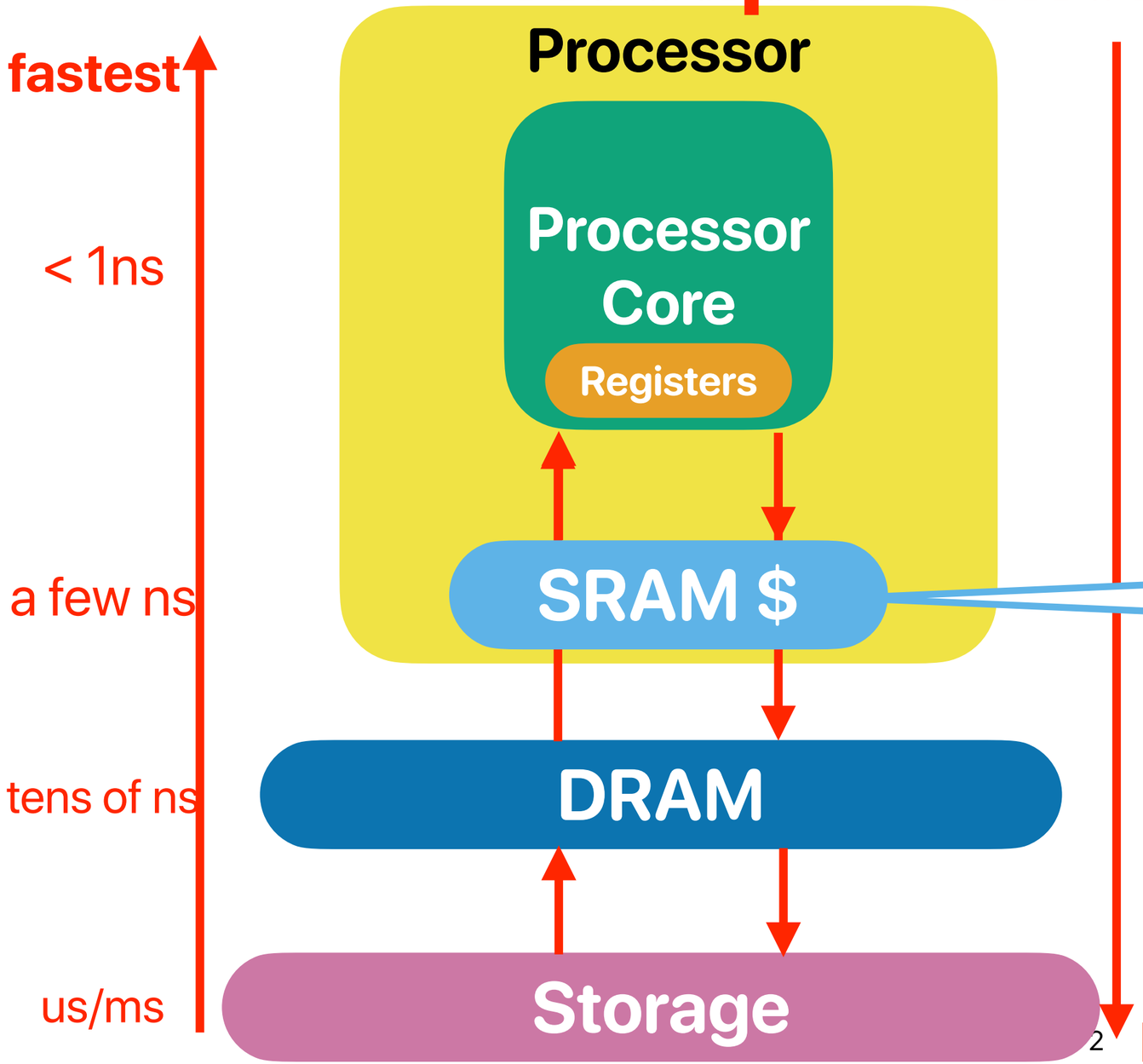
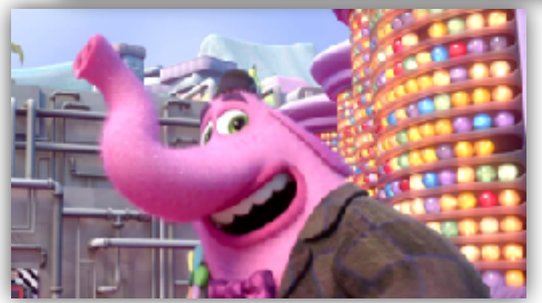
Memory hierarchy (4): cache misses and how to address them — the hardware(cont.) & software version

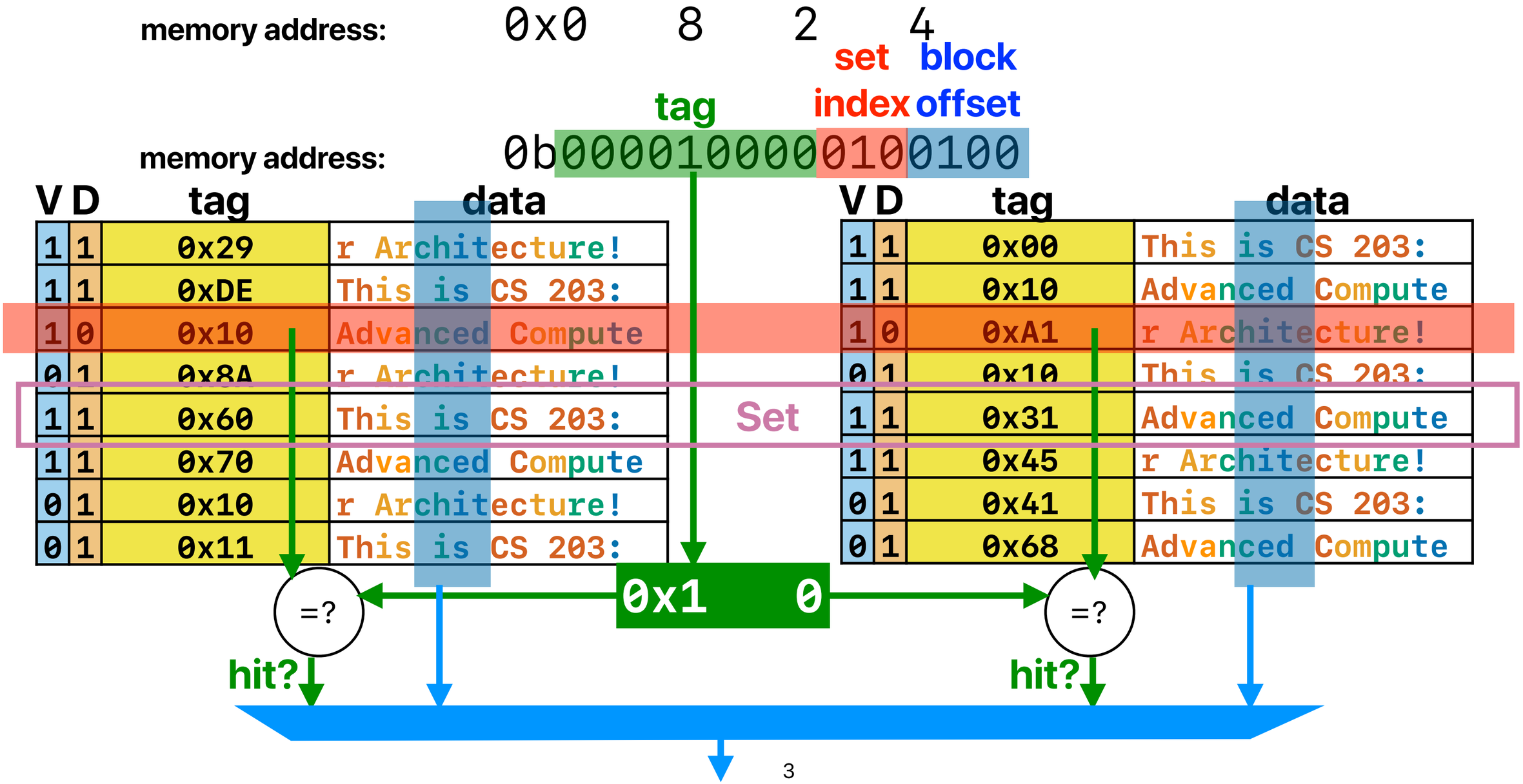
Hung-Wei Tseng

Recap: Memory Hierarchy





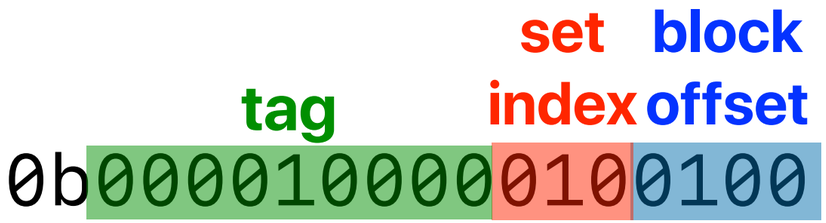
Way-associative cache



Recap: C = ABS

- **C**: **C**apacity in data arrays
- **A**: Way-**A**ssociativity — how many blocks within a set
  - N-way: N blocks in a set, A = N
  - 1 for direct-mapped cache
- **B**: **B**lock Size (Linesize)
  - How many bytes in a block
- **S**: Number of **S**ets:
  - A set contains blocks sharing the same index
  - 1 for fully associate cache
- number of bits in **b**lock offset — lg(**B**)
- number of bits in **s**et index: lg(**S**)
- tag bits: address_length - lg(S) - lg(B)
  - address_length is N bits for N-bit machines (e.g., 64-bit for 64-bit machines)
- (address / block_size) % S = set index

memory address:



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

tag **index** **offset**

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[0]	0x10000	0b000100000000000000000	0x4	0x0	Miss	
b[0]	0x20000	0b001000000000000000000	0x8	0x0	Miss	
c[0]	0x30000	0b001100000000000000000	0xC	0x0	Miss	
d[0]	0x40000	0b010000000000000000000	0x10	0x0	Miss	
e[0]	0x50000	0b010100000000000000000	0x14	0x0	Miss	a[0–7]
a[1]	0x10008	0b000100000000000001000	0x4	0x0	Miss	b[0–7]
b[1]	0x20008	0b001000000000000001000	0x8	0x0	Miss	c[0–7]
c[1]	0x30008	0b001100000000000001000	0xC	0x0	Miss	d[0–7]
d[1]	0x40008	0b010000000000000001000	0x10	0x0	Miss	e[0–7]
e[1]	0x50008	0b010100000000000001000	0x14	0x0	Miss	a[0–7]
⋮	⋮	⋮	⋮	⋮	⋮	⋮

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)**

Some common fallacies regarding assignment 2 and cache misses

- Capacity misses — it's not related to the size of the whole data structure, but the working set size and cache capacity
  - You need to demonstrate the range of data blocks touched between two observation points
  - You **always** need to show the first address and the second address indicating the observation periods
  - You **always** have to **count the number of blocks** being touched during the same period
  - You **always** have to compare the size of those blocks with your cache capacity
  - You **always** have to discuss why are they not compulsory misses
- Conflict misses
  - You **always** have to discuss why are they not compulsory misses
  - You **always** have to discuss why are they not capacity misses
    - You have to go through the same process of examining capacity misses as well
    - You have to show why those blocks are "conflicts"

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)**
- 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

Outline

- Advanced hardware techniques to optimize cache performance
- Cache-aware software optimizations

Advanced Hardware Techniques in Improving Memory Performance

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

Other hardware 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

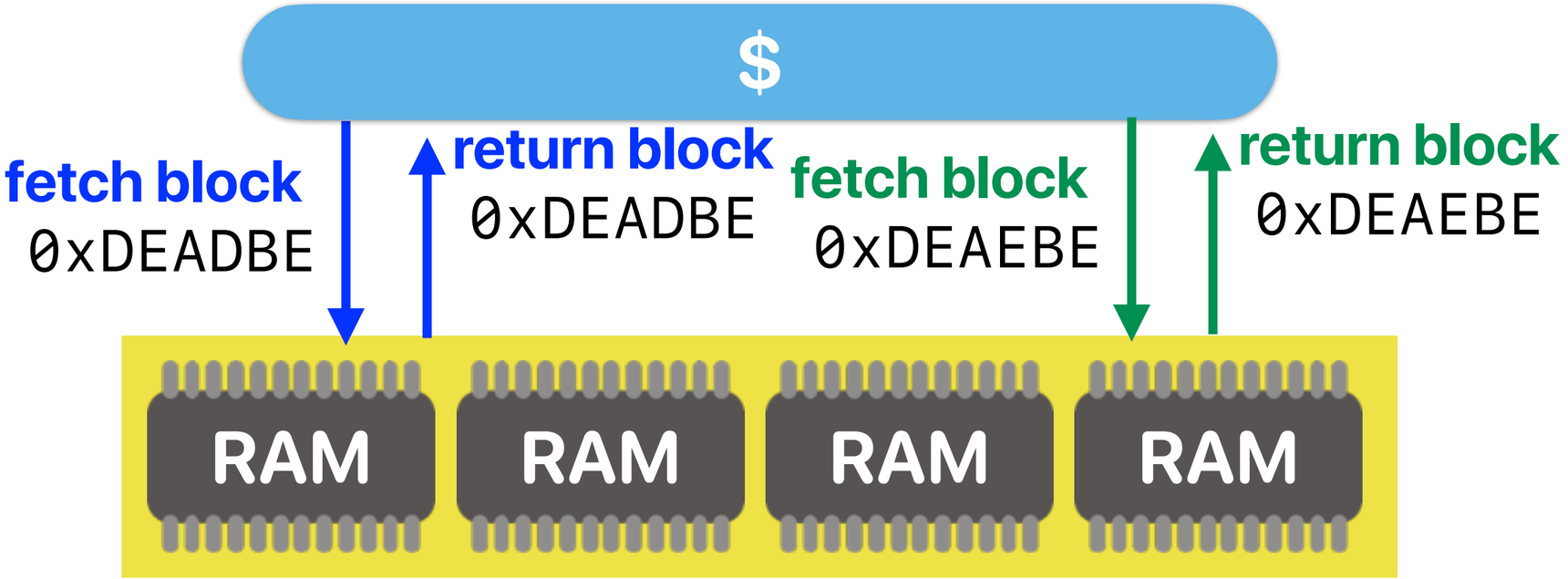
  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

Other hardware optimizations

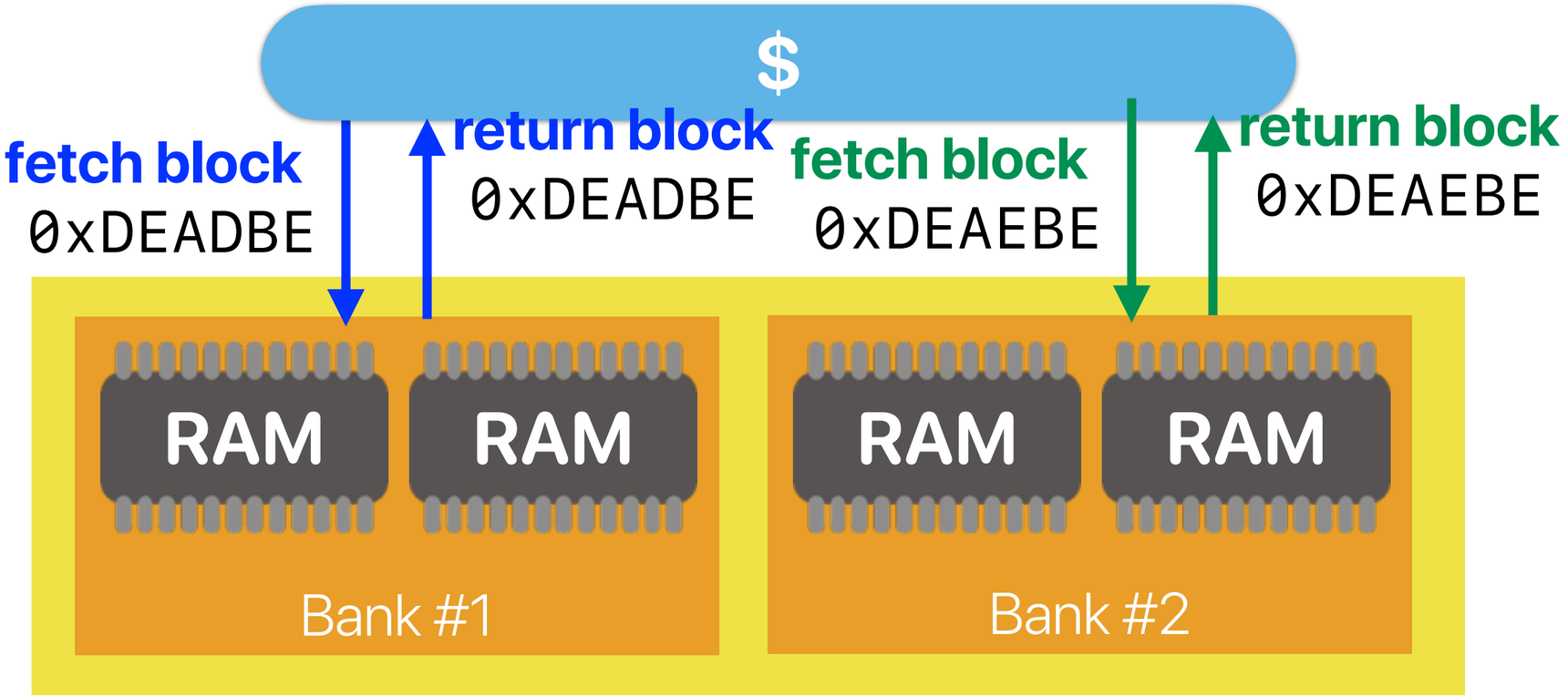
- Regarding the following cache optimizations, how many of them would help improve miss rate?
  - ① Non-blocking/pipelined/multibanked cache
  - ② Critical word first and early restart
  - ③ Prefetching
  - ④ Write buffer

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

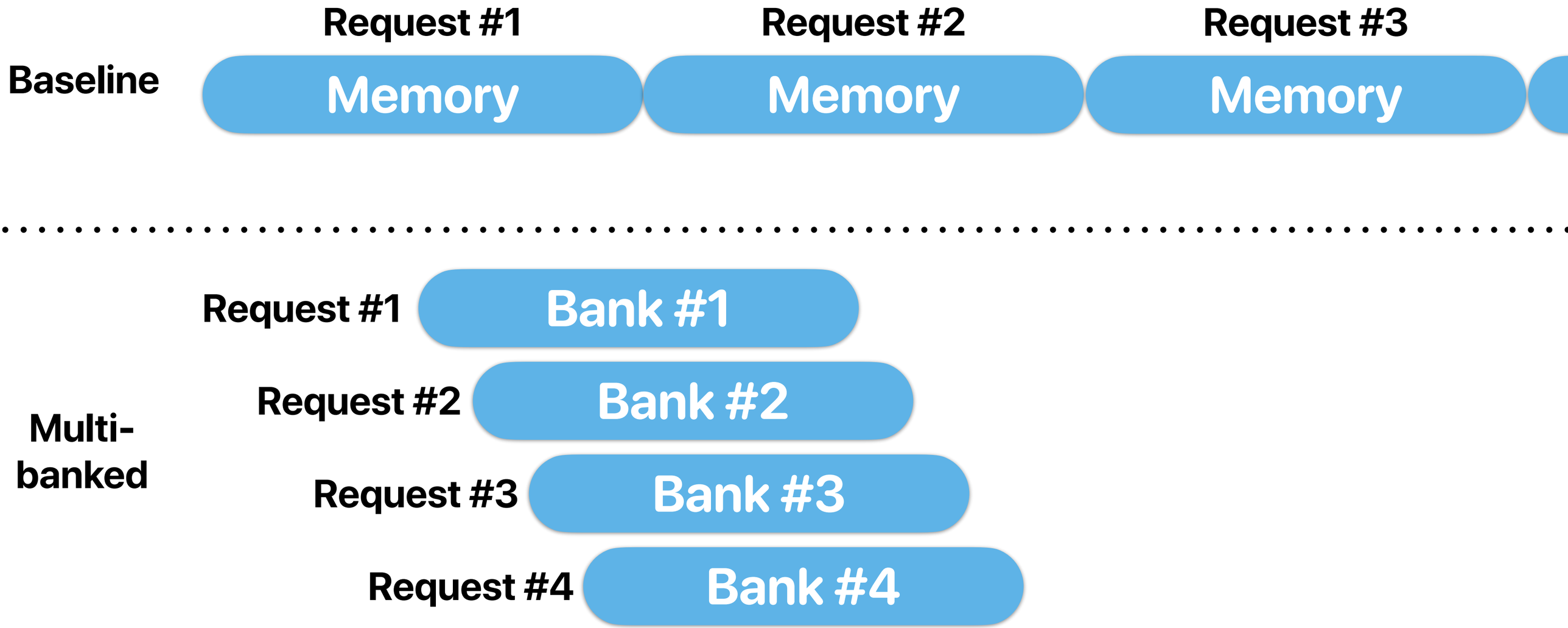
Blocking cache



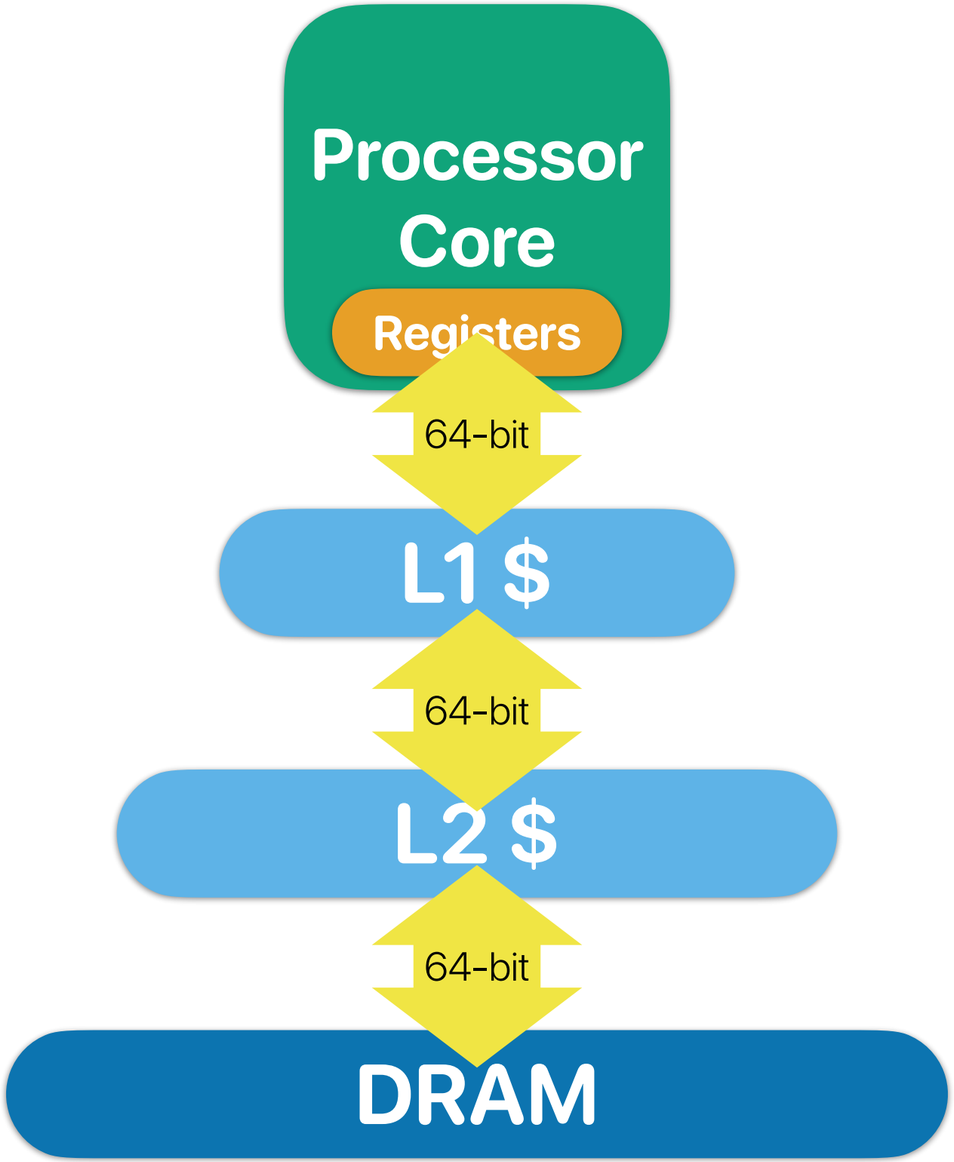
Multibanks & non-blocking caches



Pipelined access and multi-banked caches



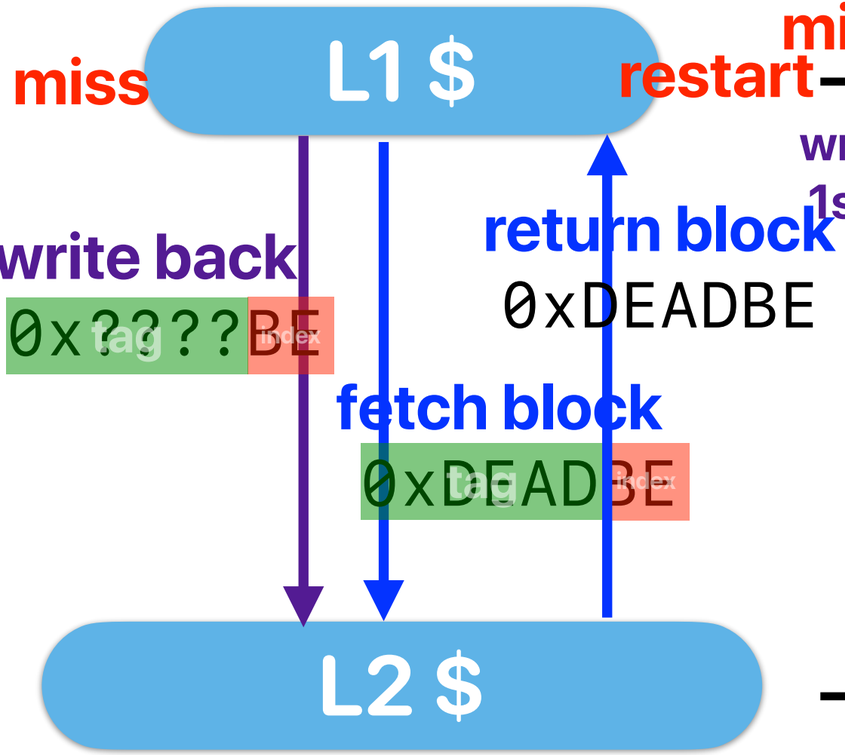
The bandwidth between units is limited

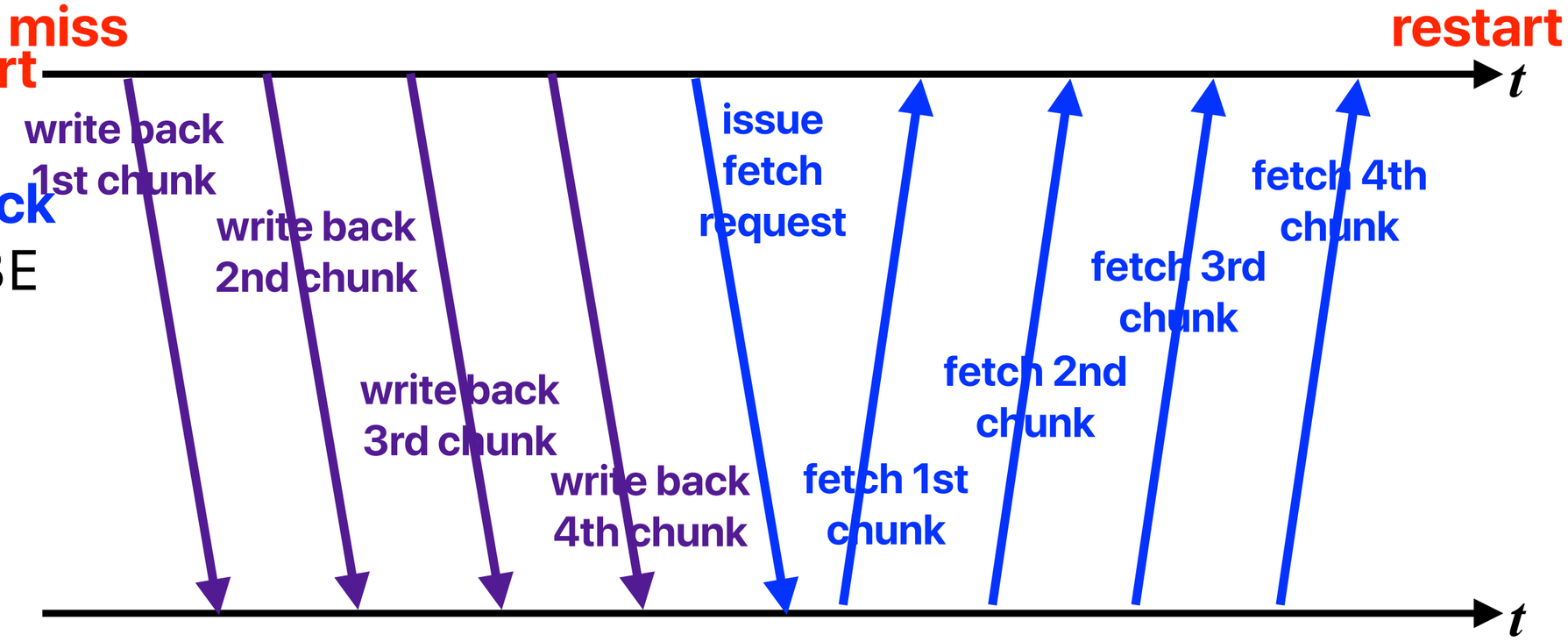


Other hardware optimizations

- 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
  - ③ Prefetching
  - ④ Write buffer
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

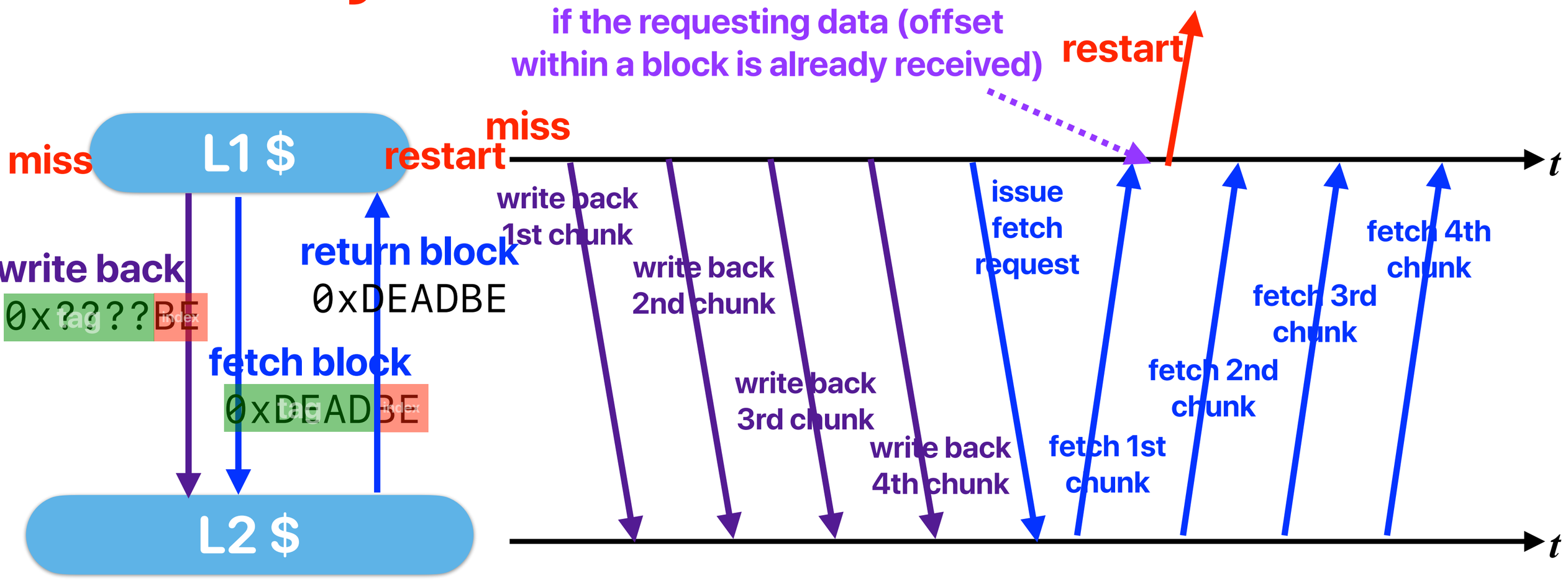
When we handle a miss





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

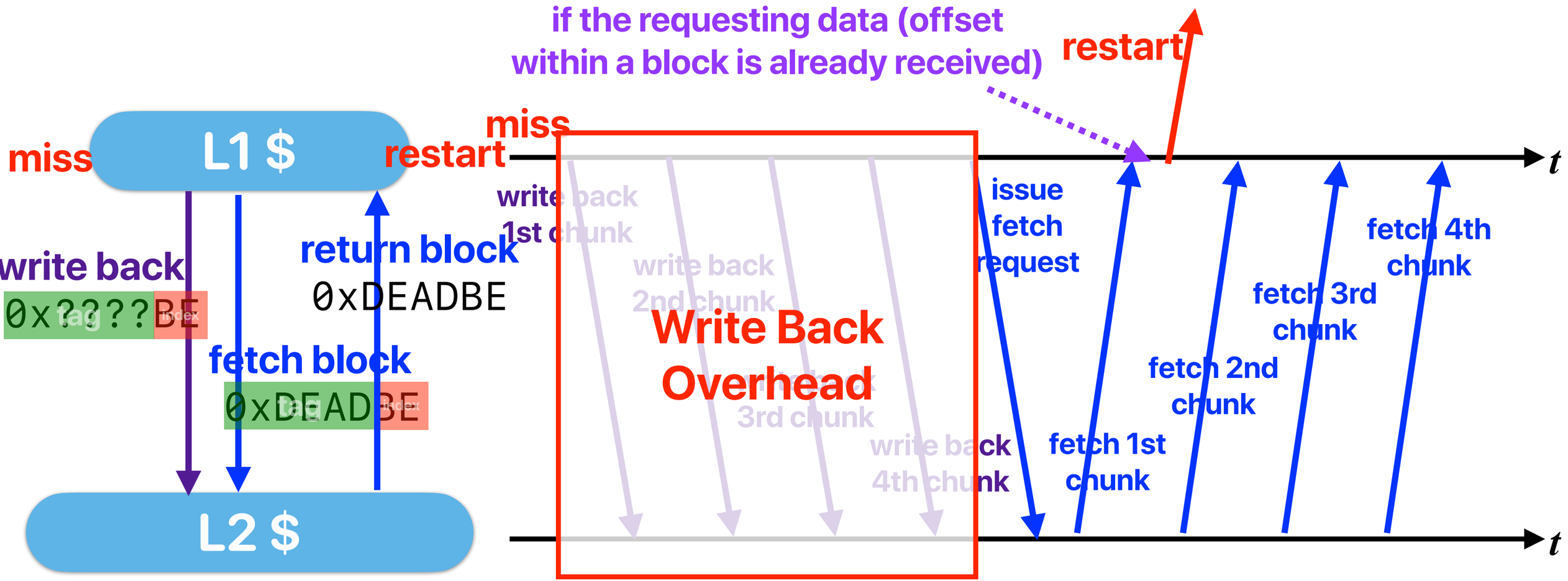
Early Restart and Critical Word First

- 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).

Other hardware optimizations

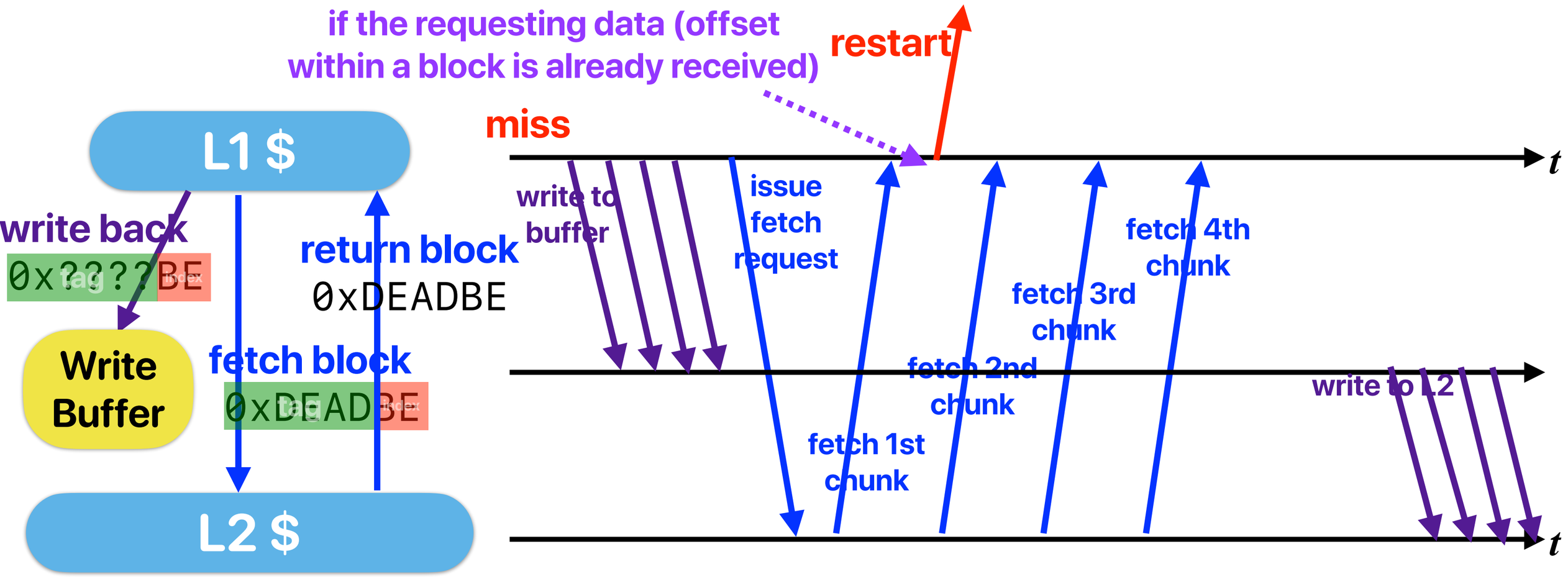
- Regarding the following cache optimizations, how many of them would help improve miss rate?
  - ① Non-blocking/pipelined/multibanked cache **Miss penalty/Bandwidth**
  - ② Critical word first and early restart **Miss penalty**
  - ③ Prefetching **Miss rate (compulsory)**
  - ④ Write buffer
  - A. 0
  - B. 1
  - C. 2
  - D. 3
  - E. 4

Can we avoid the overhead of writes?



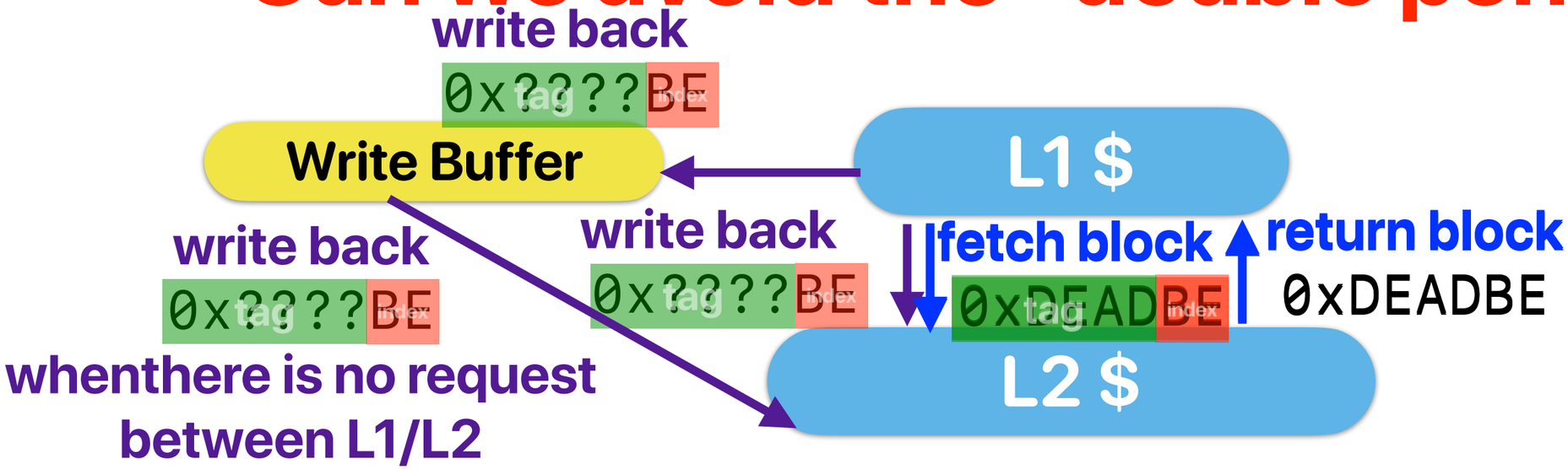
assume the bus between L1/L2 only allows a quarter of the cache block go through it

Write buffer!



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

Can we avoid the "double penalty"?



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

Can we avoid the “double penalty”?

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

Other hardware 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 Architectural Optimizations

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

How can programmer improve memory performance?

Data structures

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

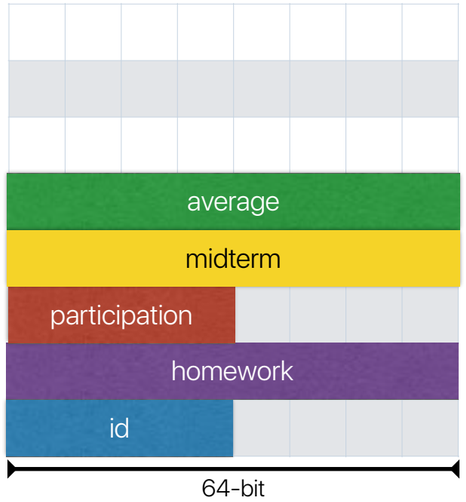
Memory addressing/alignment

- 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

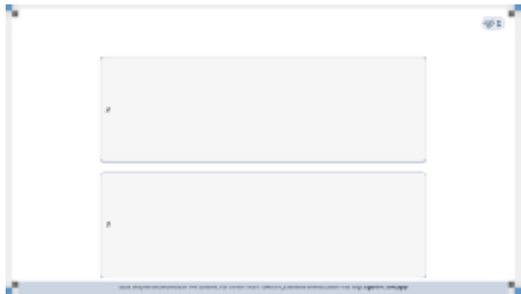
Column-store or row-store

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

A. Array of objects
B. Object of arrays



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`
	`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

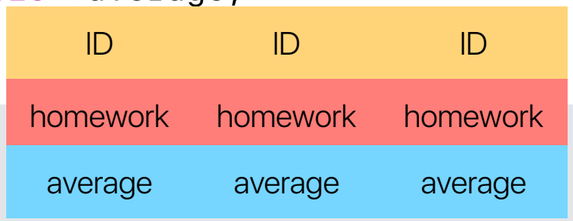
```
struct grades
{
  int id;
  double *homework;
  double average;
};
```

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;
```

Takeaways: Software Optimizations

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

Loop interchange/fission/fusion

Demo — programmer & performance

A

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

B

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

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

Loop optimizations

Loop interchange

A

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

B

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

Takeaways: Software Optimizations

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

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%

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

What if the code look like this?

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

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

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

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

What if the code look like this?

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

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

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

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

Total number of memory accesses: $4 \times 512 + 2 \times 512 = 3072$

Total number of cache misses: $4 \times \frac{512}{8} + 1 \times \frac{512}{8} = 320$

miss rate: $\frac{320}{3072} = 10.4\,\%$

Loop optimizations

Loop interchange

A

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

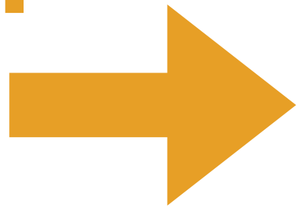
B

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

Loop fission

B

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



A

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

Takeaways: Software Optimizations

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

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];
}
```

What if we change the processor?

- If we have an intel processor with a 32KB, 8-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

Loop interchange

A

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

B

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

Loop fission

B

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

A

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

Loop fusion

A

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

B

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

Takeaways: Software Optimizations

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

Announcement

- Reading quiz #5 due next **Tuesday** before the lecture
- Assignment #2 released and due **this Thursday**
  - We do not support people on the last minute. Please carefully plan your time and respect the instructor/TA
- Midterm is using exactly the same environment and reservation system as the practice examine
  - You may start making reservations through the same link as reservation for practice examine on/after 10/23 (this Thursday)
  - First come first serve, limited slots
    - It's your responsibility to make the examine during the available period
    - If your slots are on 10/27, 10/28, you don't need to worry about assignment 3 and the lecture on 10/28 (virtual memory)
    - both versions of the midterm includes the lecture this Thursday
  - More details to come this Thursday

Some common fallacies regarding assignment 2 and cache misses

- Capacity misses — it's not related to the size of the whole data structure, but the working set size and cache capacity
  - You need to demonstrate the range of data blocks touched between two observation points
  - You **always** need to show the first address and the second address indicating the observation periods
  - You **always** have to **count the number of blocks** being touched during the same period
  - You **always** have to compare the size of those blocks with your cache capacity
  - You **always** have to discuss why are they not compulsory misses
- Conflict misses
  - You **always** have to discuss why are they not compulsory misses
  - You **always** have to discuss why are they not capacity misses
    - You have to go through the same process of examining capacity misses as well
    - You have to show why those blocks are "conflicts"

Computer
Science &
Engineering

203

つづく