Memory Hierarchy (4): Optimizing Code Performance!

Hung-Wei Tseng

Recap: Memory Hierarchy

fastest

< 1ns

a few ns

tens of ns

us/ms

Processor

Processor Core

Registers

SRAM $

DRAM

Storage

TBs

larger

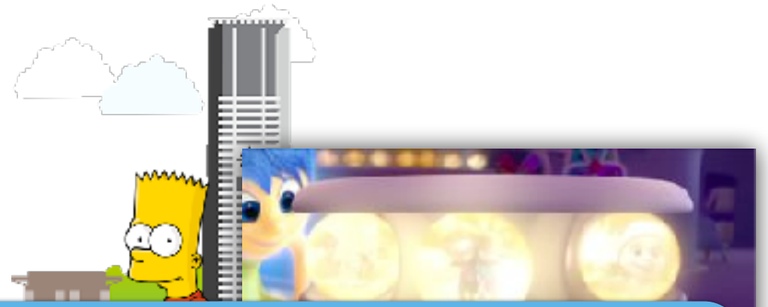
fastest

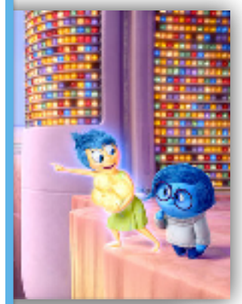
L1 $

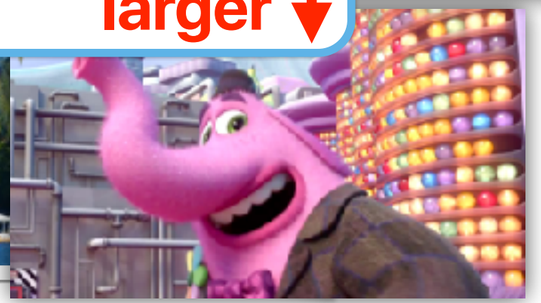
L2 $

L3 $

larger







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

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

Which of the following schemes can help Athlon 64?

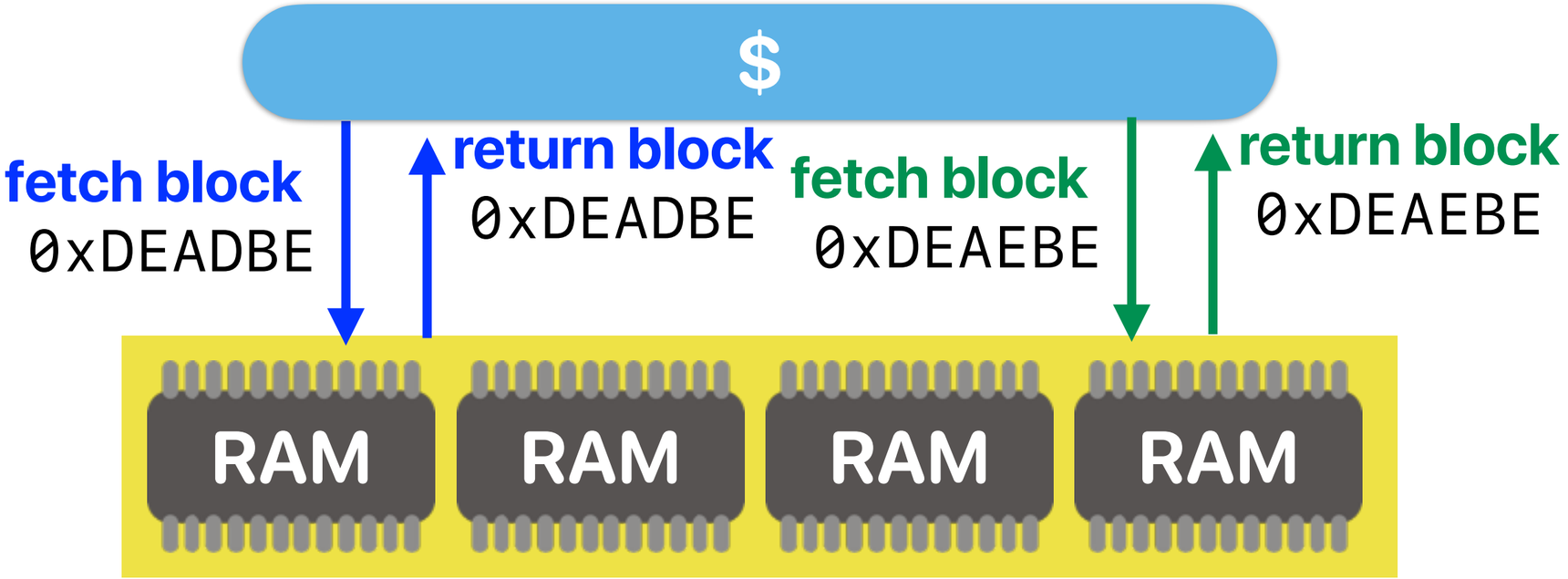
- How many of the following schemes mentioned in “improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers” would help AMD Phenom II for the code in the previous slide?
  - ① Missing cache — help improving conflict misses
  - ② Victim cache — help 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

Outline

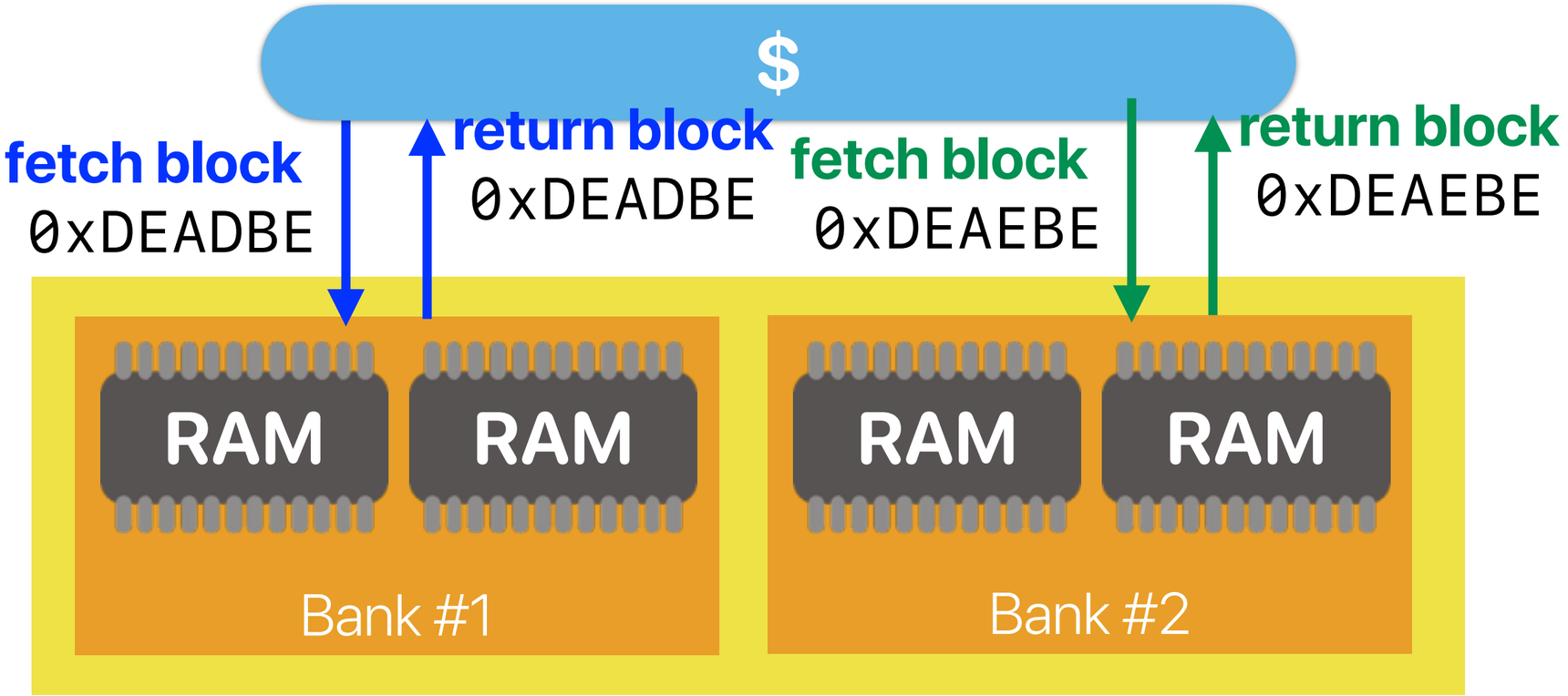
- Hardware optimizations for cache performance
- Software optimizations for cache performance

Advanced Hardware Techniques in Improving Memory Performance

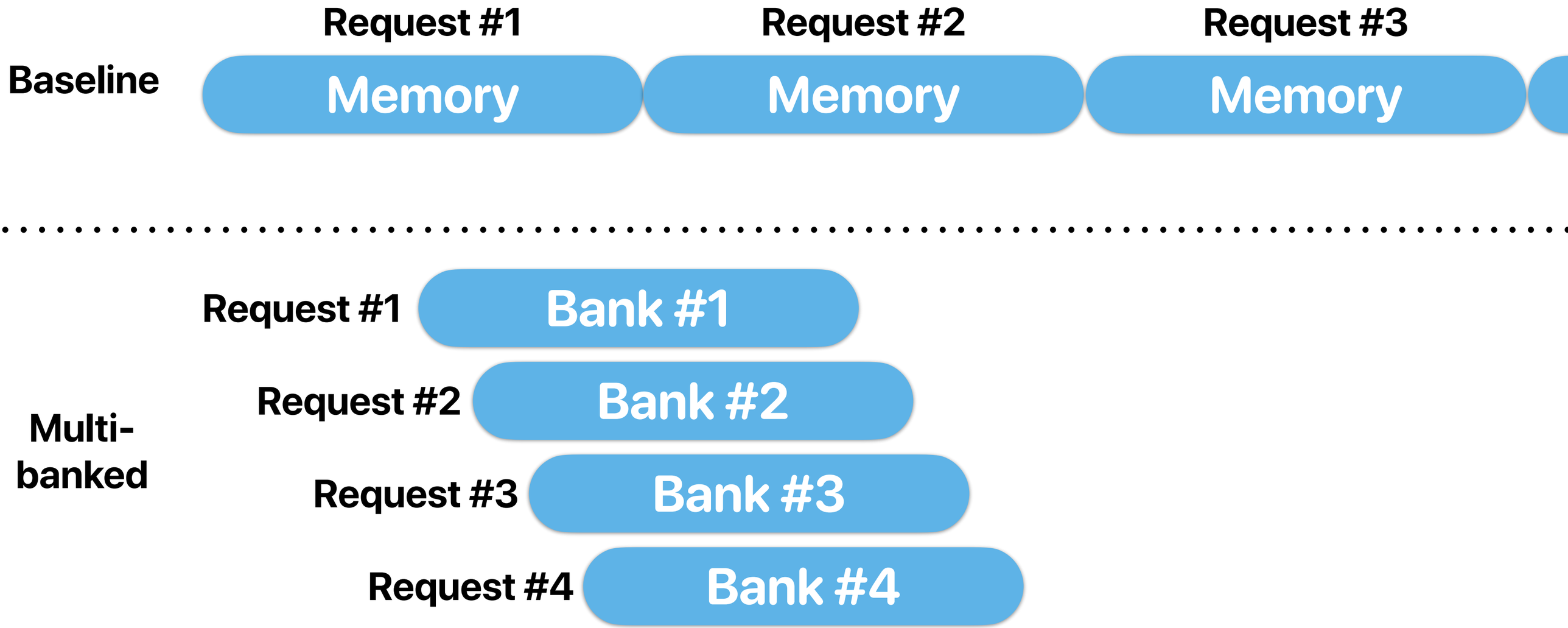
Blocking cache



Multibanks & non-blocking caches



Pipelined access and multi-banked caches



Pipelined access and multi-banked caches

- Assume each bank in the $ takes 10 ns to serve a request, and the $ can take the next request 1 ns after assigning a request to a bank — if we have 4 banks and we want to serve 4 requests, what's the speedup over non-banked, non-pipelined $? — pick the closest one
  A. 1x — no speedup
  B. 2x
  C. 3x
  D. 4x
  E. 5x

Pipelined access and multi-banked caches

- Assume each bank in the $ takes 10 ns to serve a request, and the $ can take the next request 1 ns after assigning a request to a bank — if we have 4 banks and we want to serve 4 requests, what's the speedup over non-banked, non-pipelined $? — pick the closest one

A. 1x — no speedup
B. 2x
C. 3x
D. 4x
E. 5x

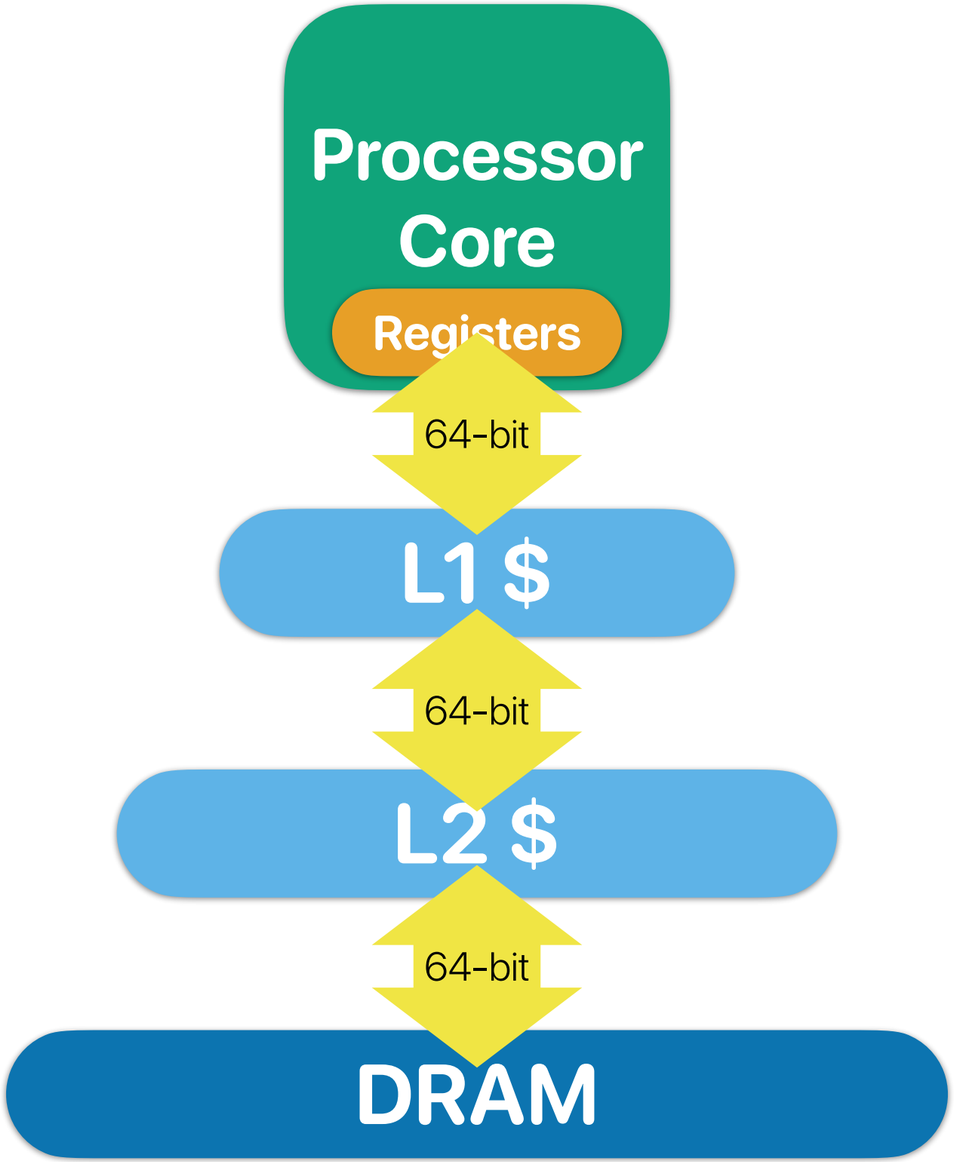
$$ET_{baseline} = 4 \times 10\ ns = 40\ ns$$

$$ET_{banked} = 10\ ns + 3 \times 1\ ns = 13\ ns$$

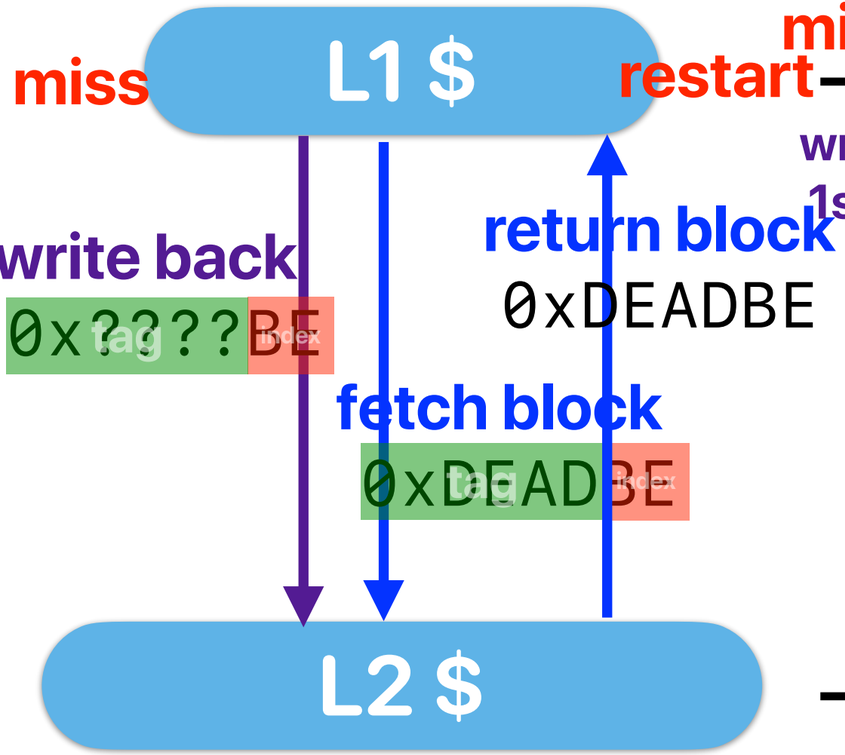
$$Speedup = \frac{Execution\ Time_{baseline}}{Execution\ Time_{banked}}$$

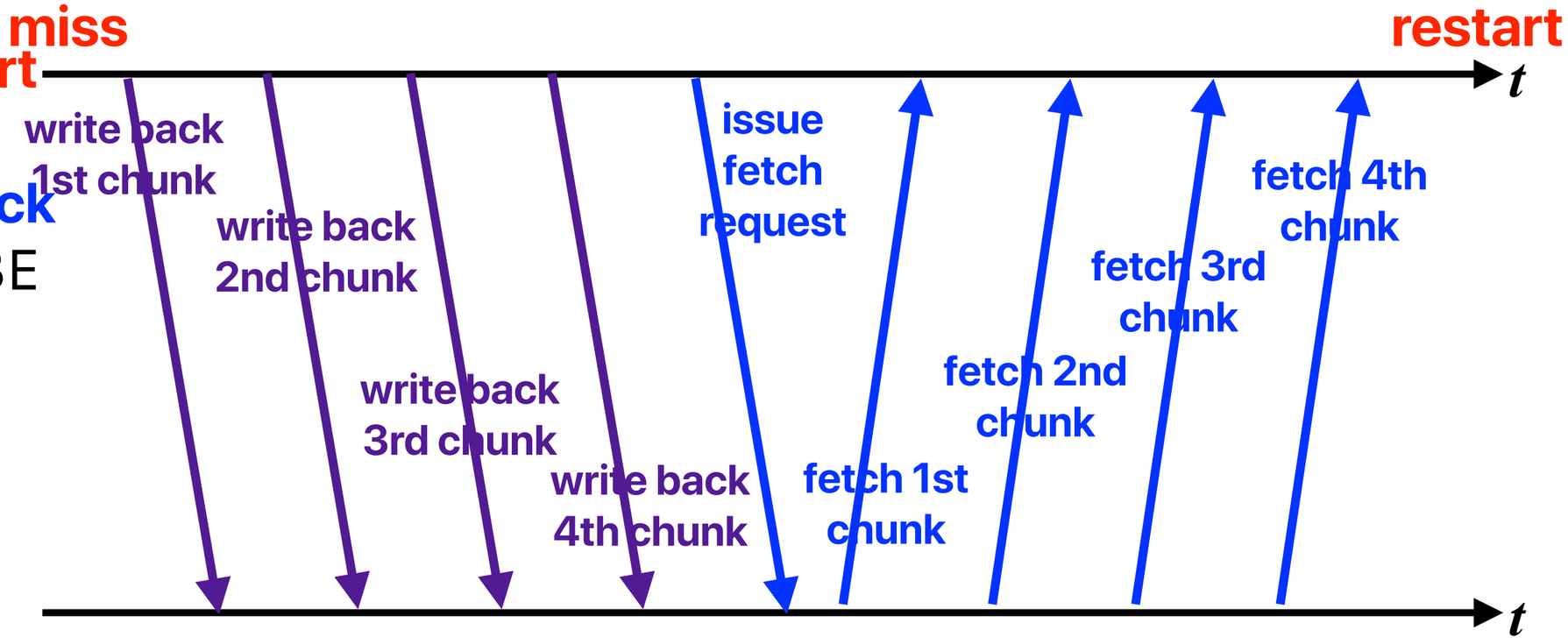
$$= \frac{40}{13} = 3.08\times$$

The bandwidth between units is limited



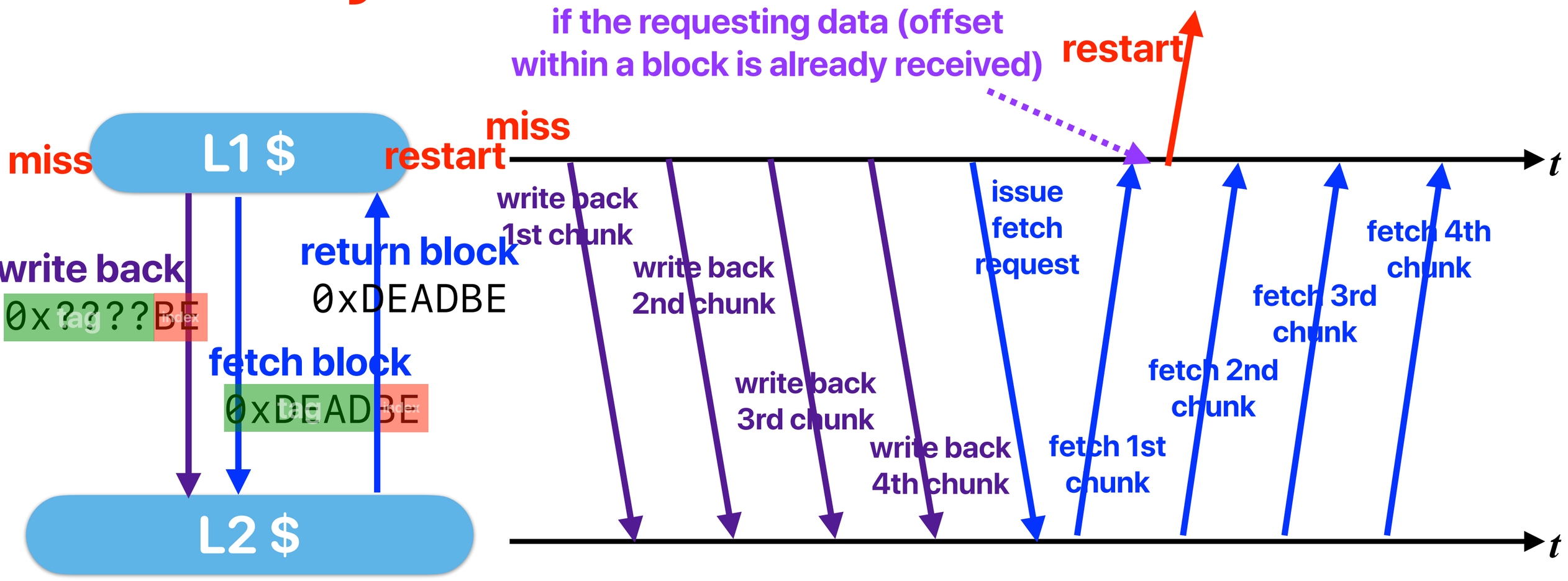
When we handle a miss





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

Early Restart and Critical Word First

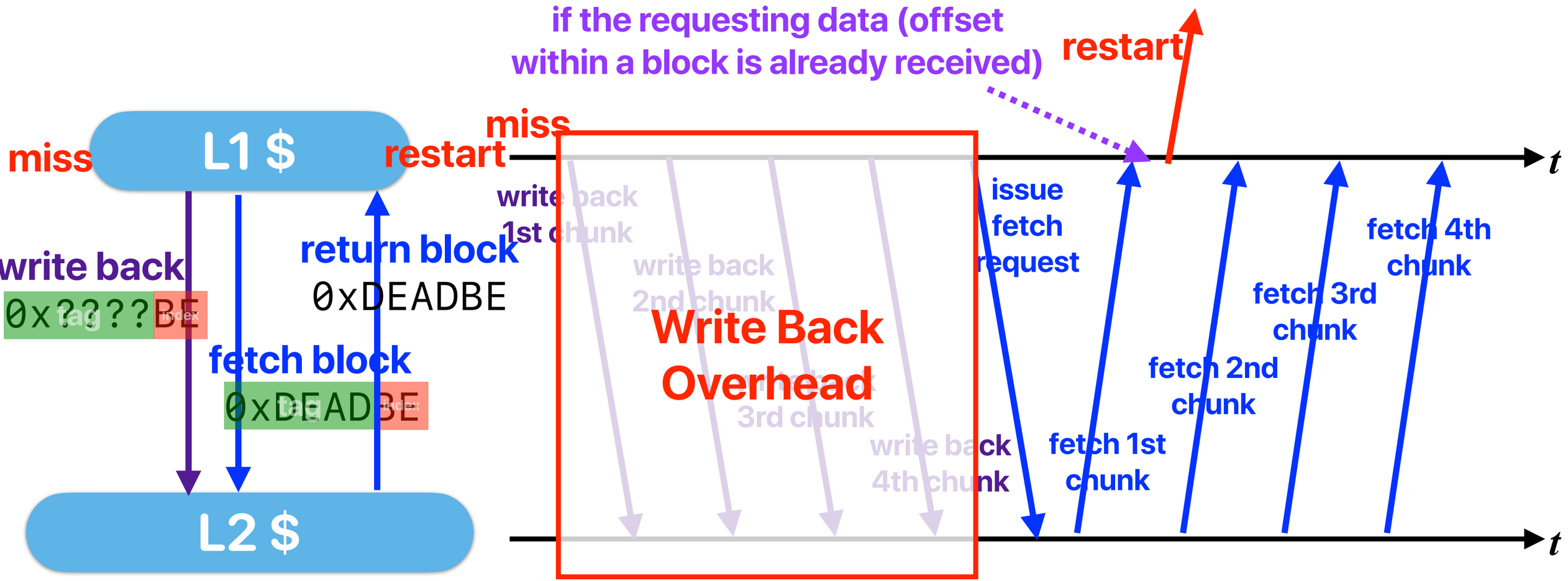


assume the bus between L1/L2 only allows a quarter of the cache block go through it

Early Restart and Critical Word First

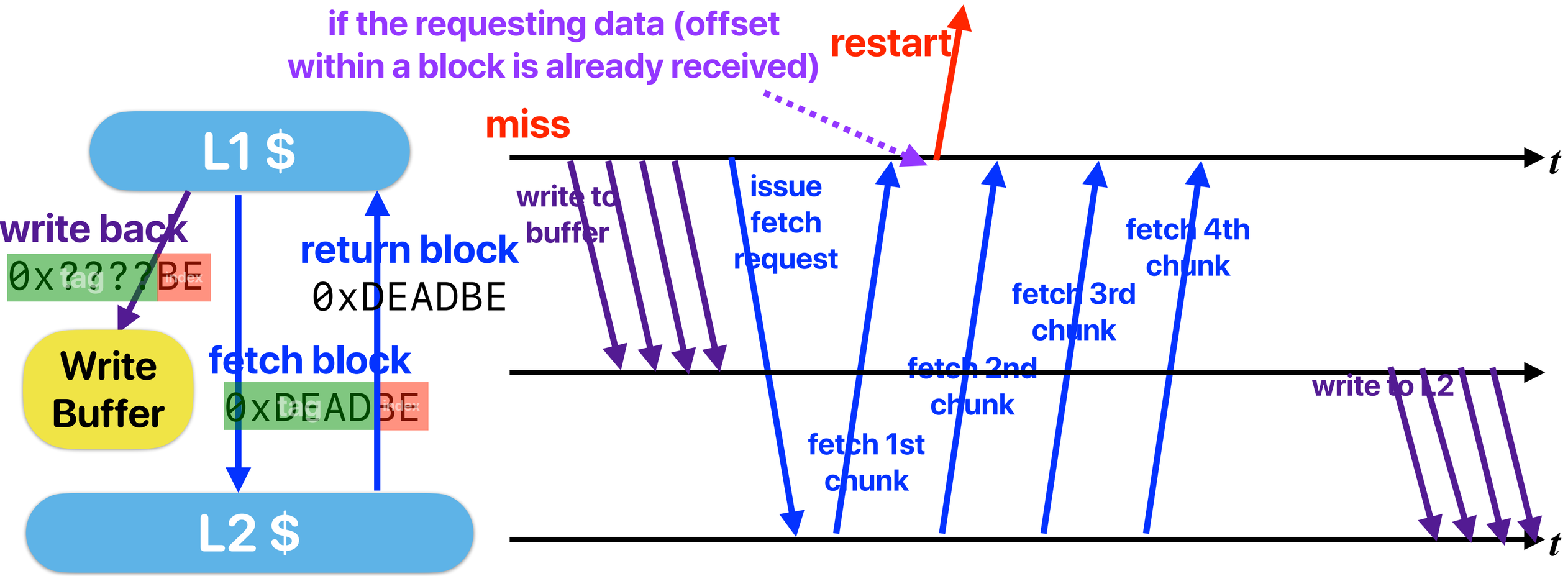
- Don’t wait for full block to be loaded before restarting CPU
  - Early restart—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
  - Critical Word First—Request the missed word first from memory and send it to the CPU as soon as it arrives; let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



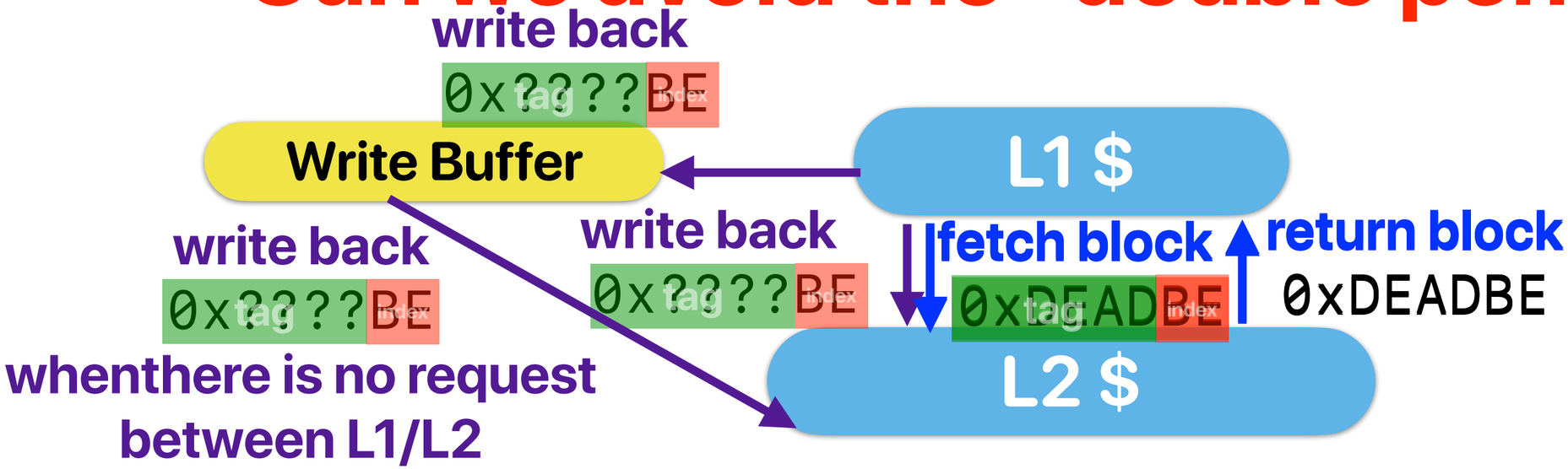
assume the bus between L1/L2 only allows a quarter of the cache block go through it

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

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

Programming and memory performance

Data layout

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

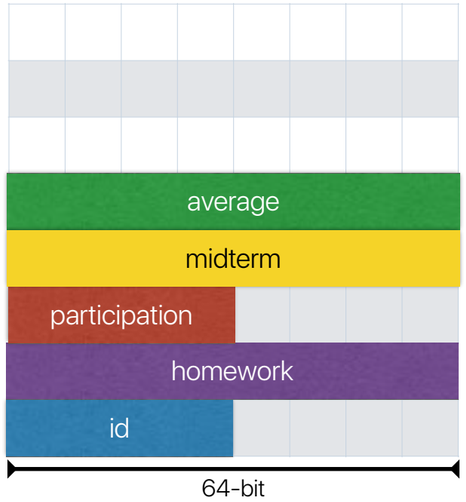
Memory addressing/alignment

- 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

Tips of software optimizations

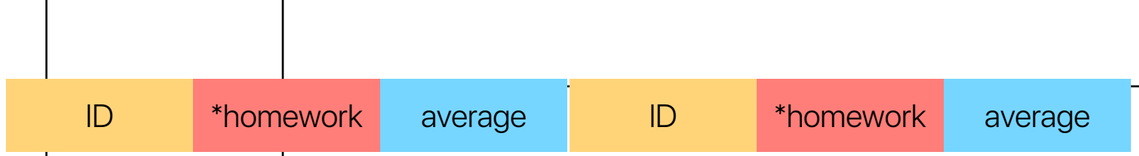
- Carefully layout your data structure can improve capacity misses!

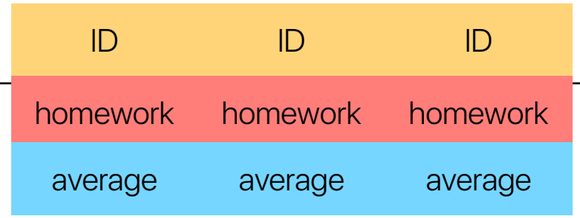
Array of structures or structure of arrays

	Array of objects	object of arrays

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





What data structure is performing better

- Considering your workload would like to calculate the average score of **one of the homework** for **all students**, which data structure would deliver better performance?
  A. Array of objects
  B. Object of arrays

	Array of objects	object of arrays

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

What data structure is performing better

	Array of objects	object of arrays

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

- Considering your workload would like to calculate the average score of **one of the homework** for **all students**, which data structure would deliver better performance? **What if we want to calculate average scores for each student?**

  A. Array of objects

  B. Object of arrays

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;
```

Tips of software optimizations

- Carefully layout your data structure can improve capacity misses!
- Make your data structures align with the access pattern can better exploit cache locality — improve conflict misses

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

AMD Phenom II

- D-L1 Cache configuration of AMD Phenom II
  - Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 32-bit address.

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

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

A. 6.25%
B. 56.25%
C. 66.67%
D. 68.75%
E. 100%

C = ABS

64KB = 2 * 64 * S

S = 512

offset = lg(64) = 6 bits

index = lg(512) = 9 bits

tag = 64 - lg(512) - lg(64) = 49 bits

What if the code look like this?

- D-L1 Cache configuration of AMD Phenom II
  - Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 32-bit address.

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

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

A. 5%
B. 6.25%
C. 66.67%
D. 68.75%
E. 93.75%

AMD Phenom II

- Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

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

C = ABS
64KB = 2 * 64 * S
S = 512
offset = lg(64) = 6 bits
index = lg(512) = 9 bits
tag = the rest bits

	address in hex	address in binary (tag / index / offset)	tag	index	hit? miss?
load a[0]	0x20000	0b10 0000 0000 0000 0000	0x4	0	miss
store c[0]	0x10000	0b01 0000 0000 0000 0000	0x2	0	miss
load a[1]	0x20004	0b10 0000 0000 0000 0100	0x4	0	hit
store c[1]	0x10004	0b01 0000 0000 0000 0100	0x2	0	hit
⋮	⋮	⋮	⋮	⋮	⋮
load a[16]	0x20040	0b10 0000 0000 0100 0000	0x4	1	miss
store c[16]	0x10040	0b01 0000 0000 0100 0000	0x2	1	miss
⋮	⋮	⋮	⋮	⋮	⋮
load b[0]	0x30000	0b11 0000 0000 0000 0000	0x6	0	miss
load c[0]	0x10000	0b01 0000 0000 0000 0000	0x2	0	miss
store c[0]	0x10000	0b01 0000 0000 0000 0000	0x2	0	hit

512×2 **accesses**
$\frac{512}{16} \times 2$ **misses**
512×3 **accesses**

total 512×5 **accesses** **total** $\frac{512}{16} \times 4$ **misses** $\frac{512}{16} \times 2$ **misses**

What if the code look like this?

- D-L1 Cache configuration of AMD Phenom II
  - Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 32-bit address.

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

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

A. 5%
B. 6.25%
C. 66.67%
D. 68.75%
E. 93.75%

$$\frac{\frac{512}{16} \times 4}{512 \times 5} = 0.05$$

Loop fission

Loop Fusion

```
/* Before */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        d[i][j] = a[i][j] + c[i][j];


/* After */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
    {
        a[i][j] = 1/b[i][j] * c[i][j];
         d[i][j] = a[i][j] + c[i][j];
    }
```

2 misses per access to a & c vs. one miss per access

Tips of software optimizations

- Carefully layout your data structure can improve capacity misses!
- Make your data structures align with the access pattern can better exploit cache locality — improve conflict misses
- Implementing algorithms in a more cache friendly way!

Announcement

- Reading Quiz #5 next Monday before the lecture
- Assignment #2 due next Wednesday midnight — will be up later today
- Office Hours
  - Walk-in, no appointment is necessary
  - Hung-Wei/Prof. Usagi: MTu 2p-3p (WCH 406 or on Zoom)
  - Abenezer Wudenhe: WTh 3p-4p (Zoom only)