

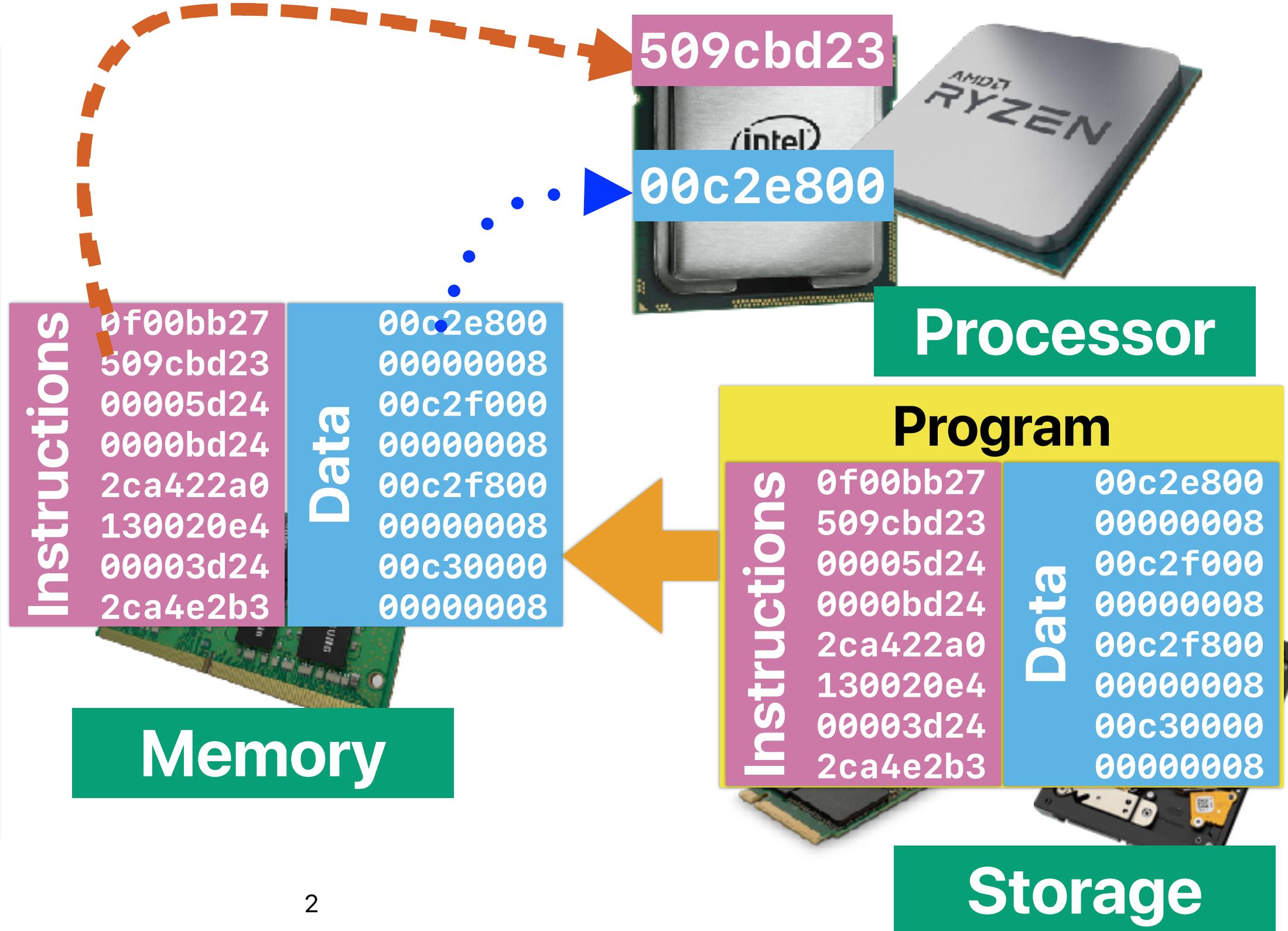
Memory Hierarchy Inside Out:

(4) Cache misses and their remedies

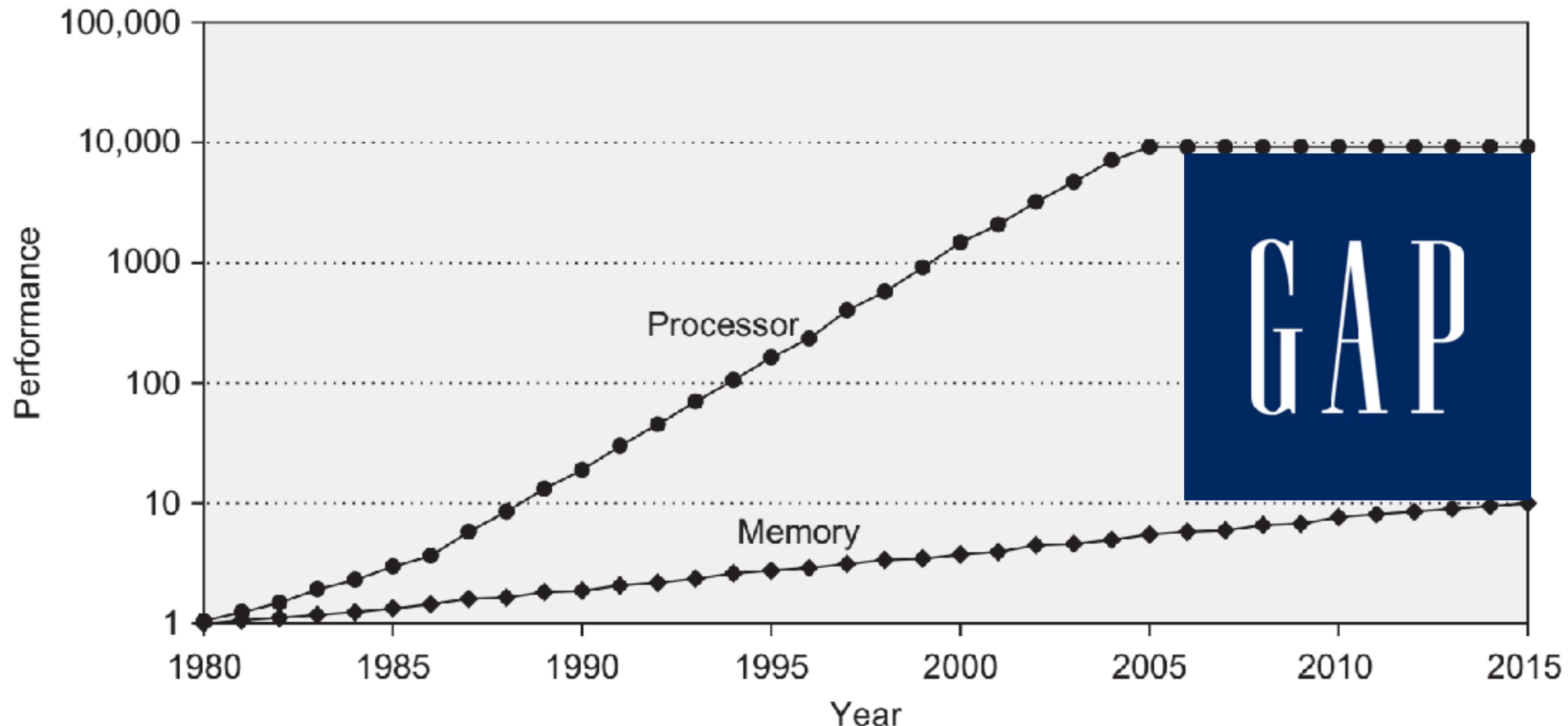
— the software version

Hung-Wei Tseng

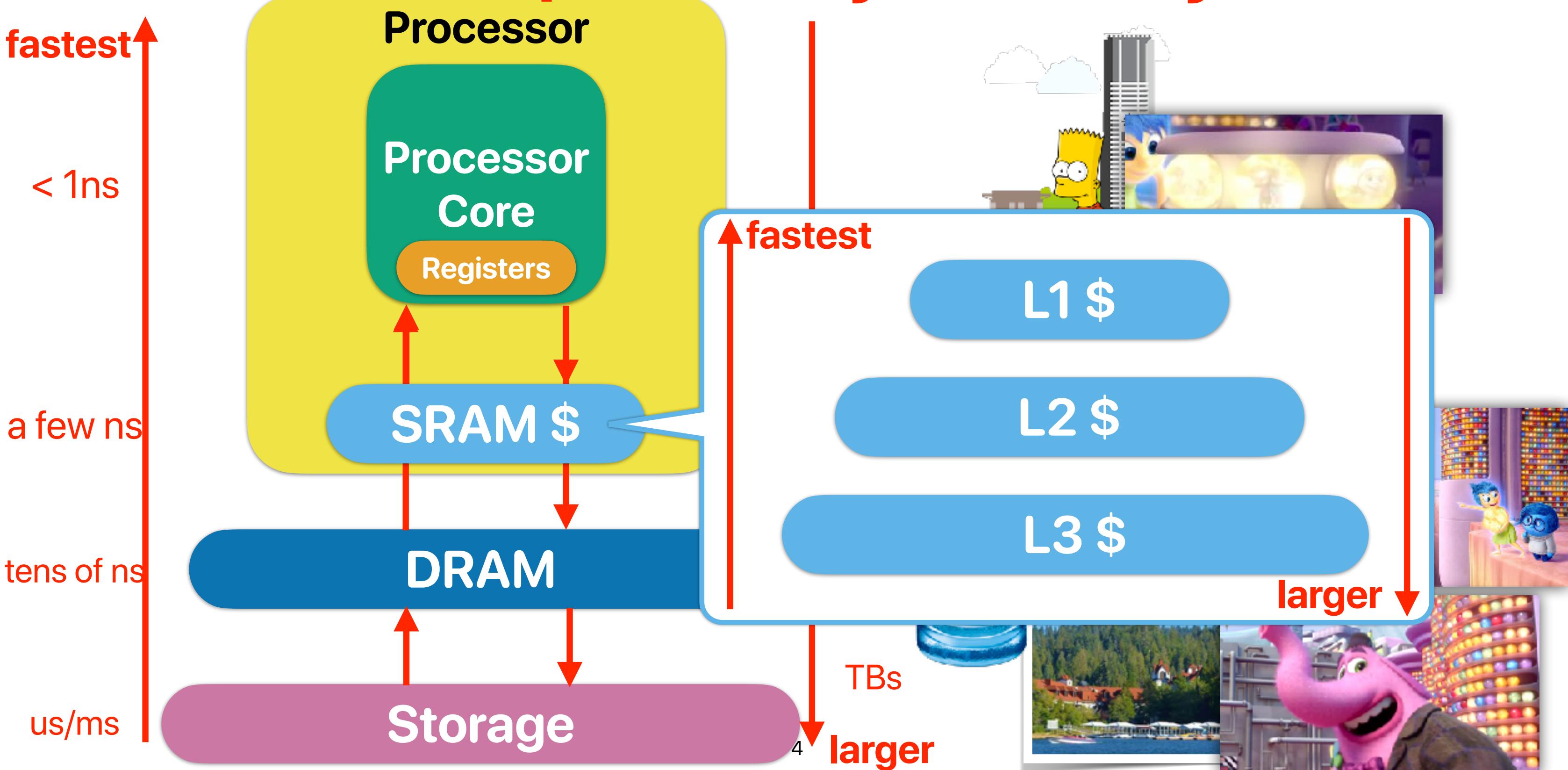
von Neumann Architecture



Recap: Performance gap between Processor/Memory



Recap: Memory Hierarchy



Review: C = ABS

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

memory address:



Review: 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

Recap: 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[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]
    tag index offset
```

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

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?
a[0]	0x10000	0b000100000000000000000000000000	0x8	0x0	Compulsory Miss	
b[0]	0x20000	0b001000000000000000000000000000	0x10	0x0	Compulsory Miss	
c[0]	0x30000	0b001100000000000000000000000000	0x18	0x0	Compulsory Miss	
d[0]	0x40000	0b010000000000000000000000000000	0x20	0x0	Compulsory Miss	
e[0]	0x50000	0b010100000000000000000000000000	0x28	0x0	Compulsory Miss	a[0-7]
a[1]	0x10008	0b000100000000000000100000000000	0x8	0x0	Conflict Miss	b[0-7]
b[1]	0x20008	0b001000000000000000000000001000	0x10	0x0	Conflict Miss	c[0-7]
c[1]	0x30008	0b001100000000000000000000001000	0x18	0x0	Conflict Miss	d[0-7]
d[1]	0x40008	0b010000000000000000000000001000	0x20	0x0	Conflict Miss	e[0-7]
e[1]	0x50008	0b010100000000000000000000001000	0x28	0x0	Conflict Miss	a[0-7]

Improving Direct-Mapped Cache Performance by the Addition of a Small Fully- Associative Cache and Prefetch Buffers

Norman P. Jouppi



Review: 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

Recap: NVIDIA Tegra X1 with prefetch

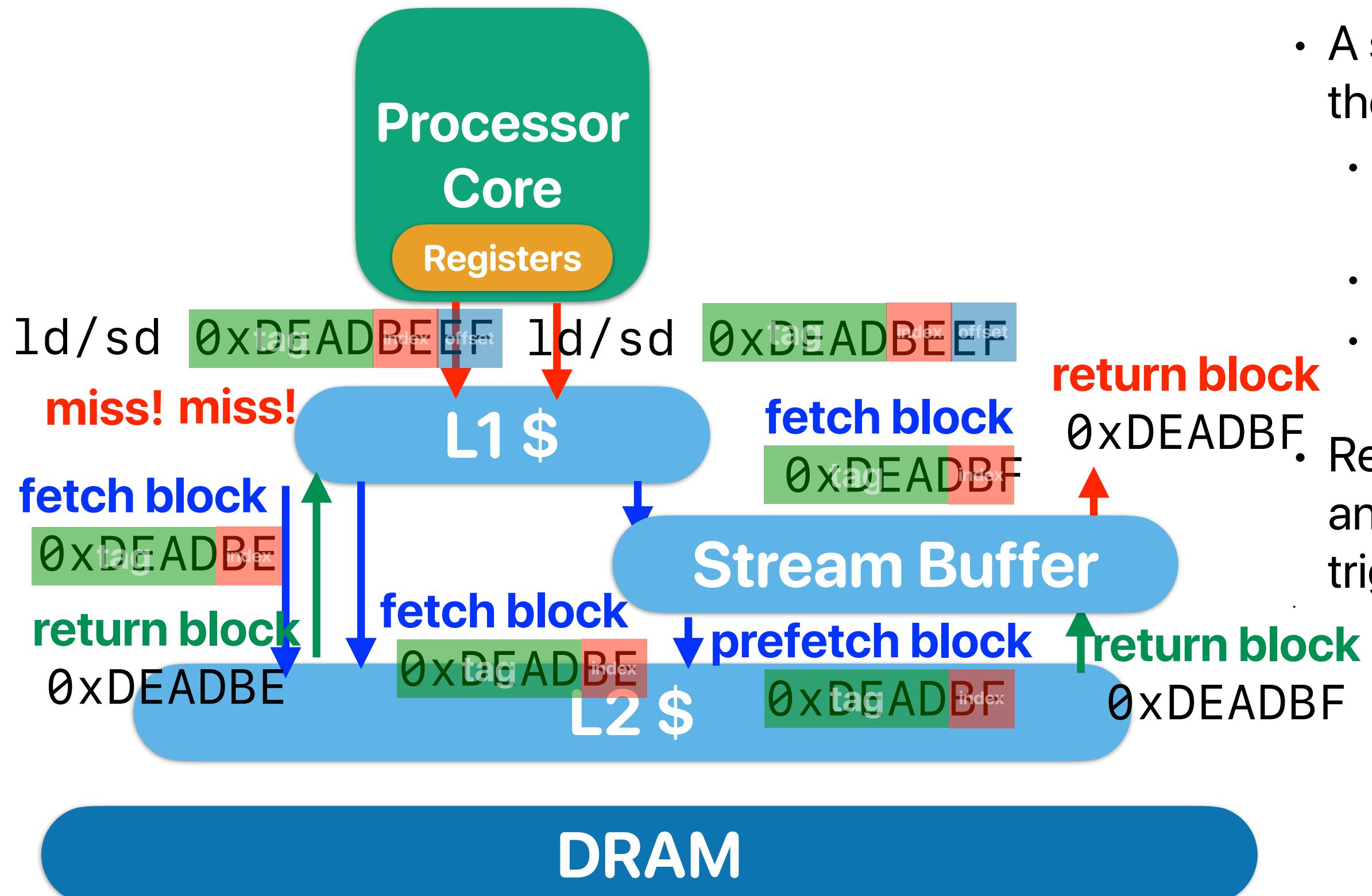
- Size 32KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
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]
```

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

	Address (Hex)	Address in binary	Tag	Index	Hit? Miss?	Replace?	Prefetch
a[0]	0x10000	0b000100000000000000000000000000	0x8	0x0	Miss		a[8-15]
b[0]	0x20000	0b001000000000000000000000000000	0x10	0x0	Miss		b[8-15]
c[0]	0x30000	0b001100000000000000000000000000	0x18	0x0	Miss		c[8-15]
d[0]	0x40000	0b010000000000000000000000000000	0x20	0x0	Miss		d[8-15]
e[0]	0x50000	0b010100000000000000000000000000	0x28	0x0	Miss	a[0-7]	e[8-15]
a[1]	0x10008	0b0001000000000000000000001000	0x8	0x0	Miss	b[0-7]	e[8-15] will kick out a[8-15]
b[1]	0x20008	0b0010000000000000000000001000	0x10	0x0	Miss	c[0-7]	
c[1]	0x30008	0b0011000000000000000000001000	0x18	0x0	Miss	d[0-7]	
d[1]	0x40008	0b0100000000000000000000001000	0x20	0x0	Miss	e[0-7]	
e[1]	0x50008	0b0101000000000000000000001000	0x28	0x0	Miss	a[0-7]	

Recap: 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

Which of the following schemes can help Tegra?

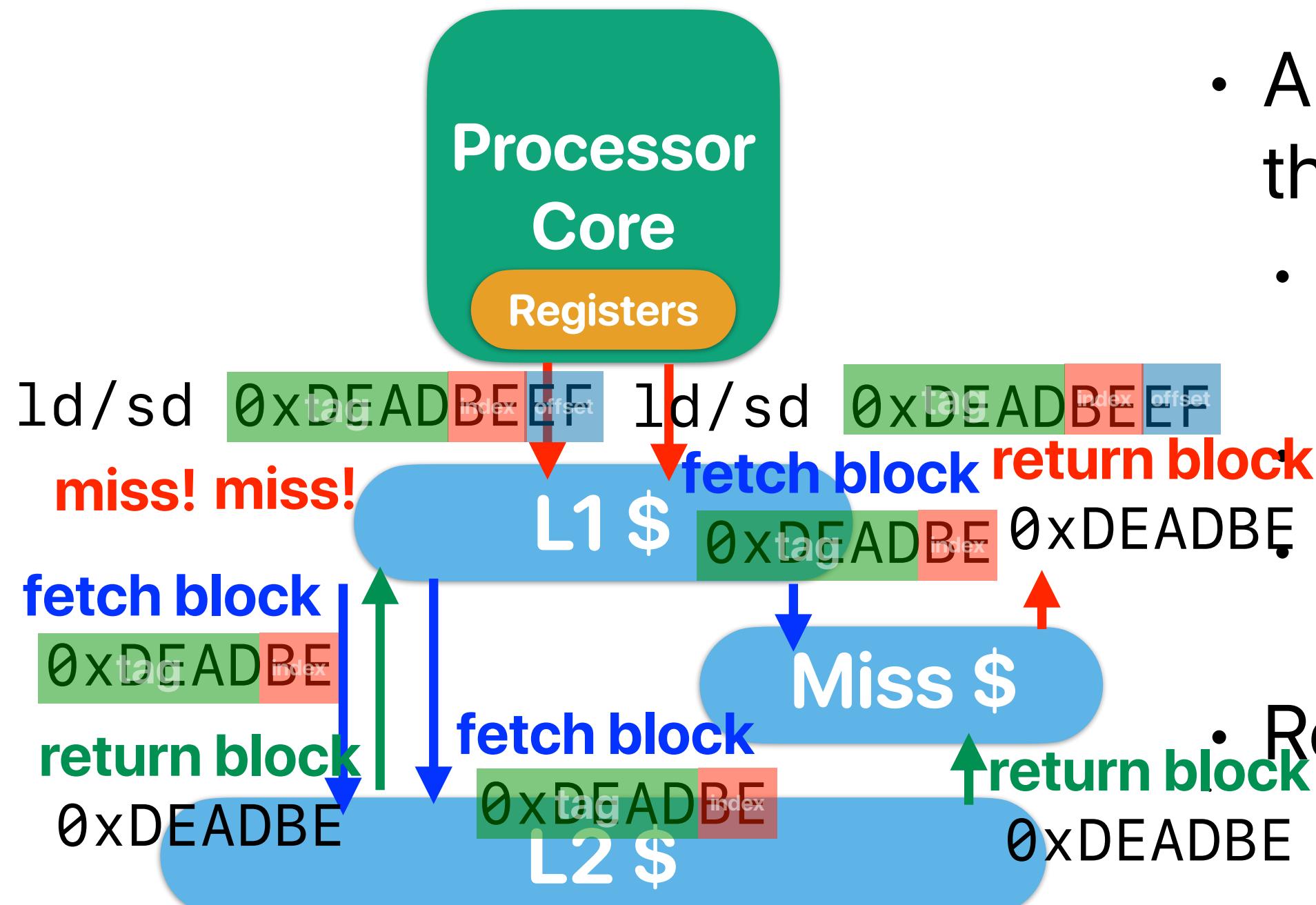
- How many of the following schemes mentioned in “improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers” would help NVIDIA’s Tegra for the code in the previous slide?
 - ① Missing cache
 - ② Victim cache
 - ③ Prefetch
 - ④ Stream buffer
- A. 0
B. 1
C. 2
D. 3
E. 4



Outline

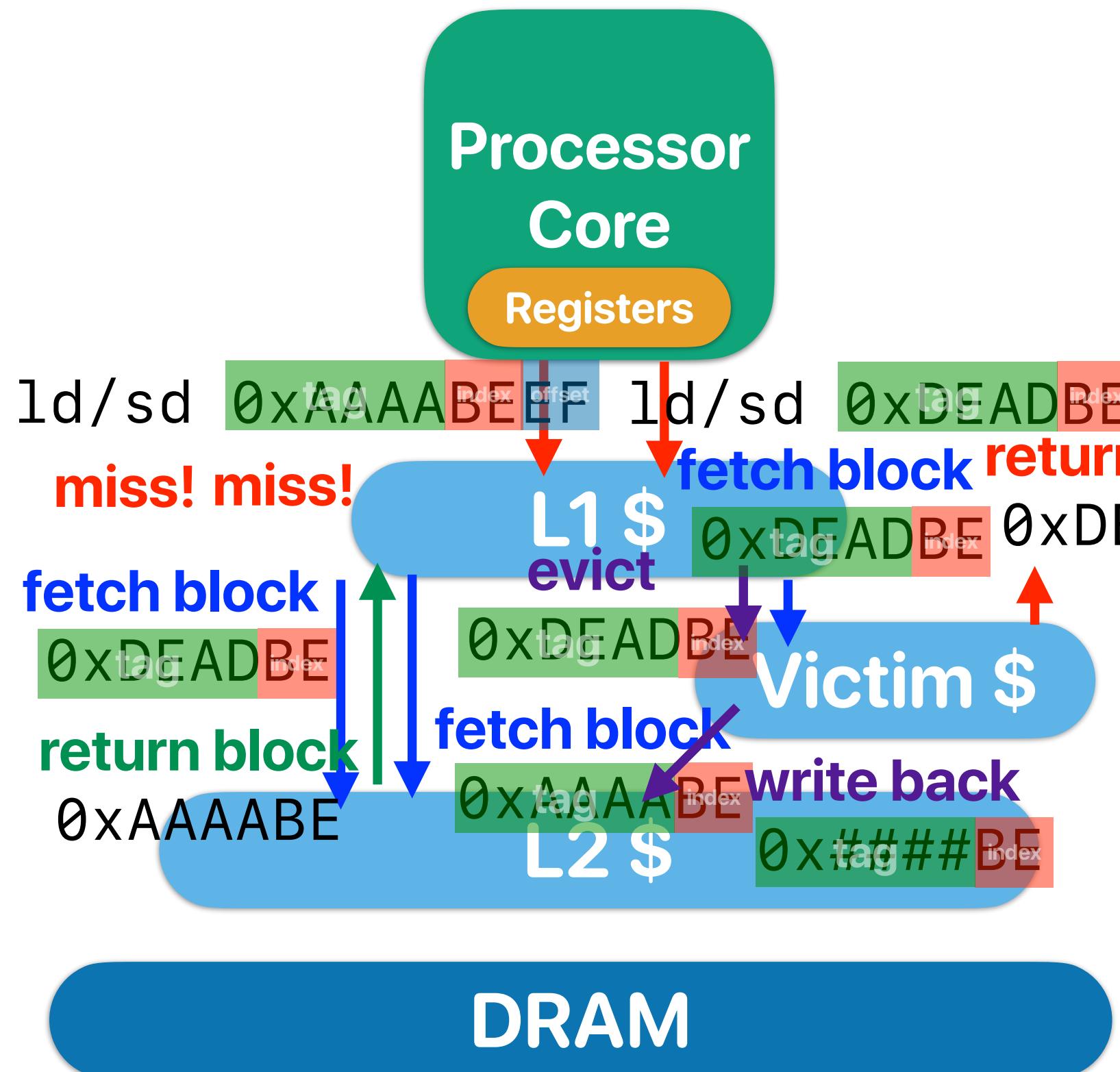
- The remedies of cache misses — the hardware version
- The remedies of cache misses — the software version

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

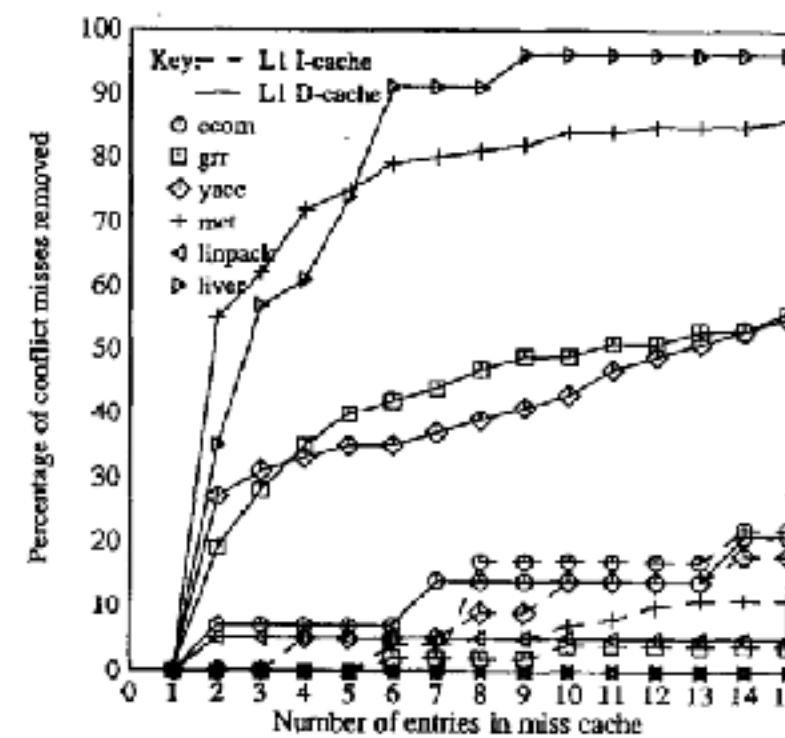


Figure 3-3: Conflict misses removed by miss caching

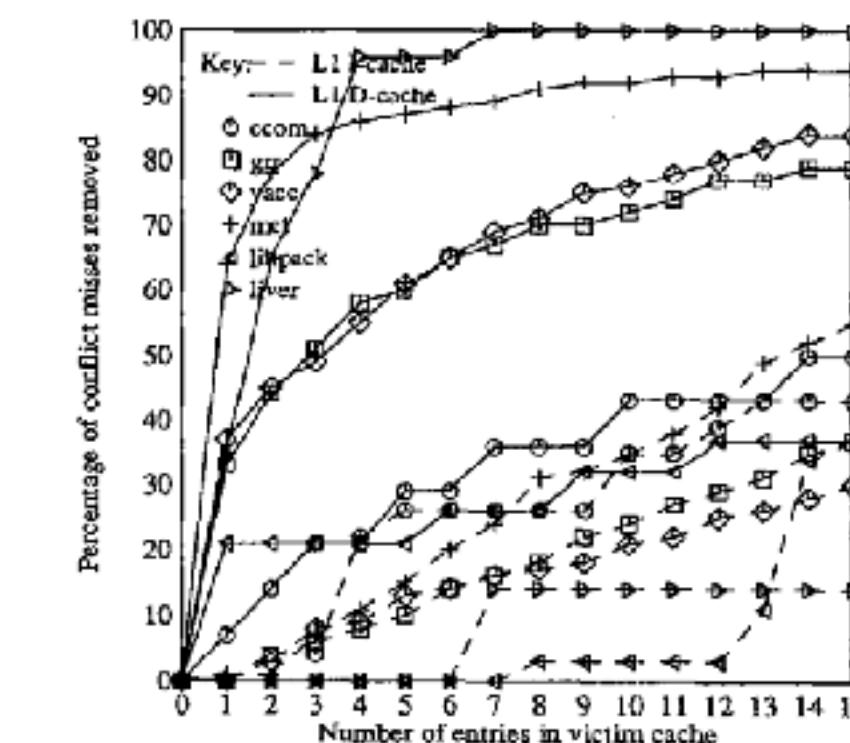


Figure 3-5: Conflict misses removed by victim caching

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

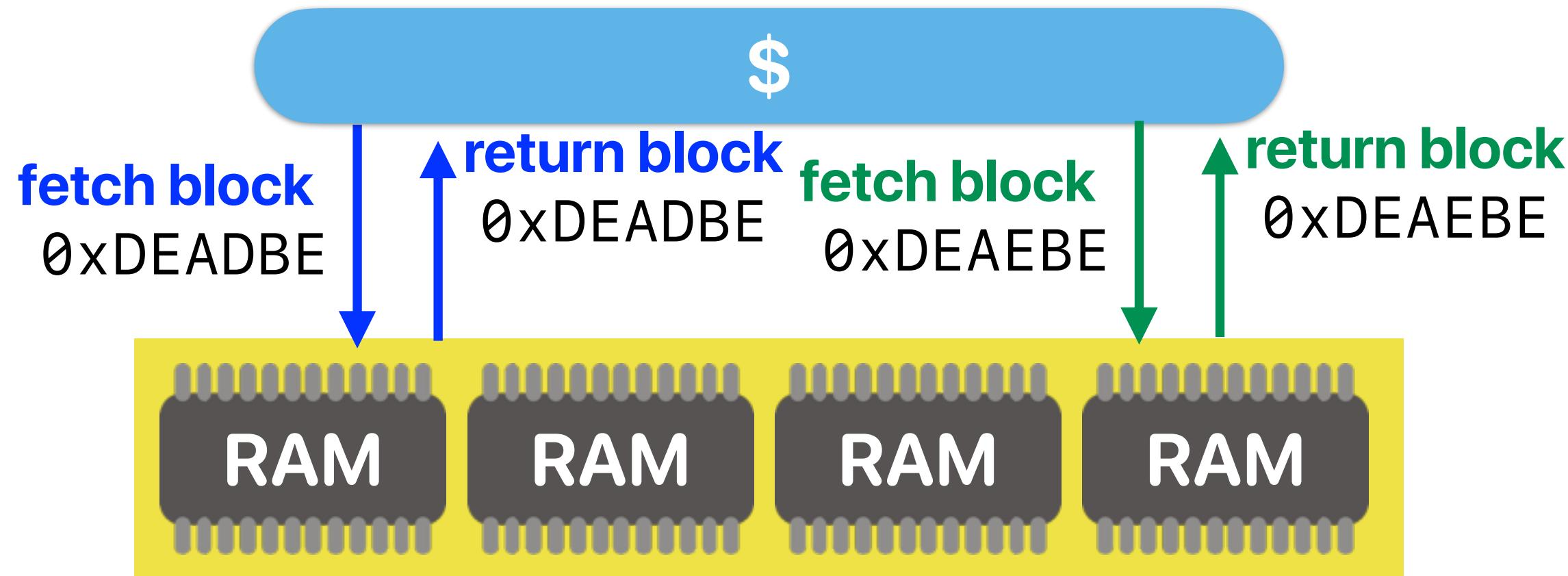
C. 2

D. 3

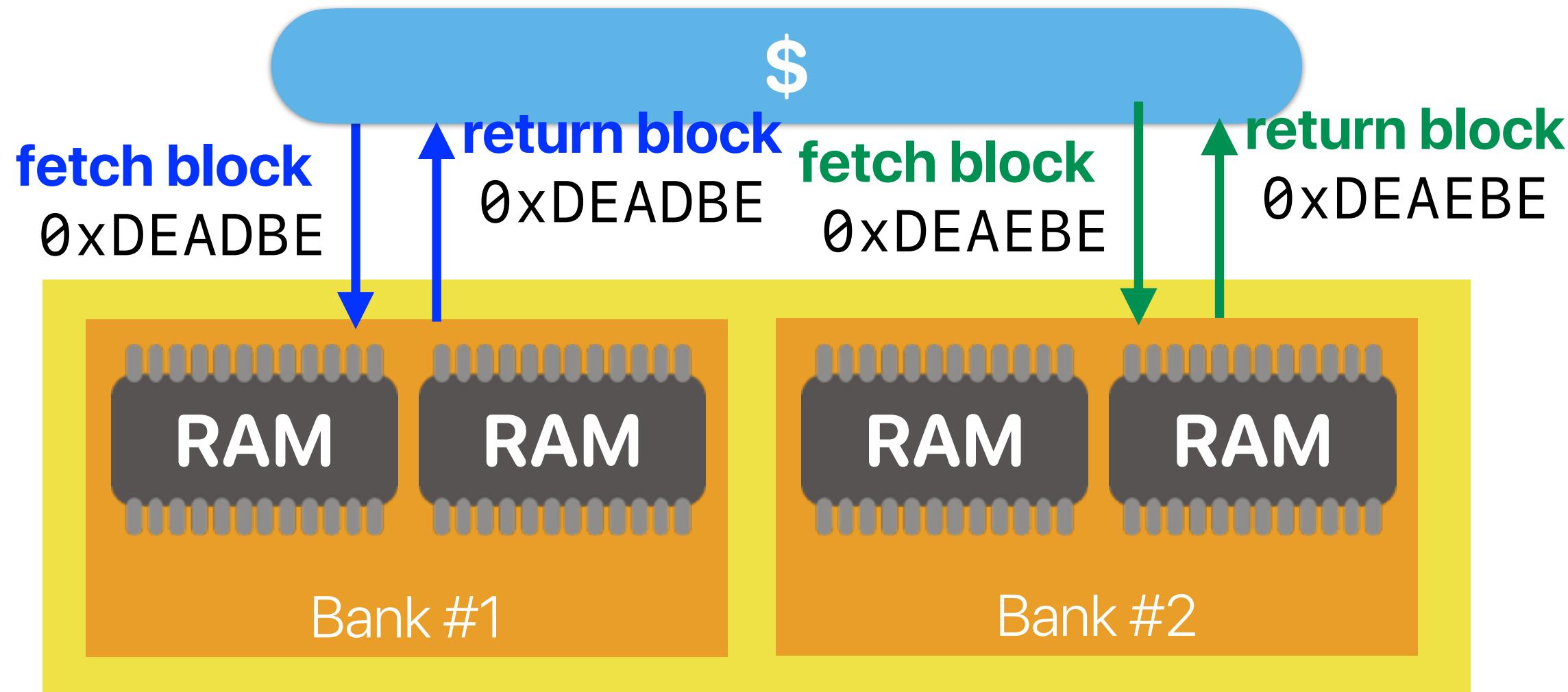
E. 4

Advanced Hardware Techniques in Improving Memory Performance

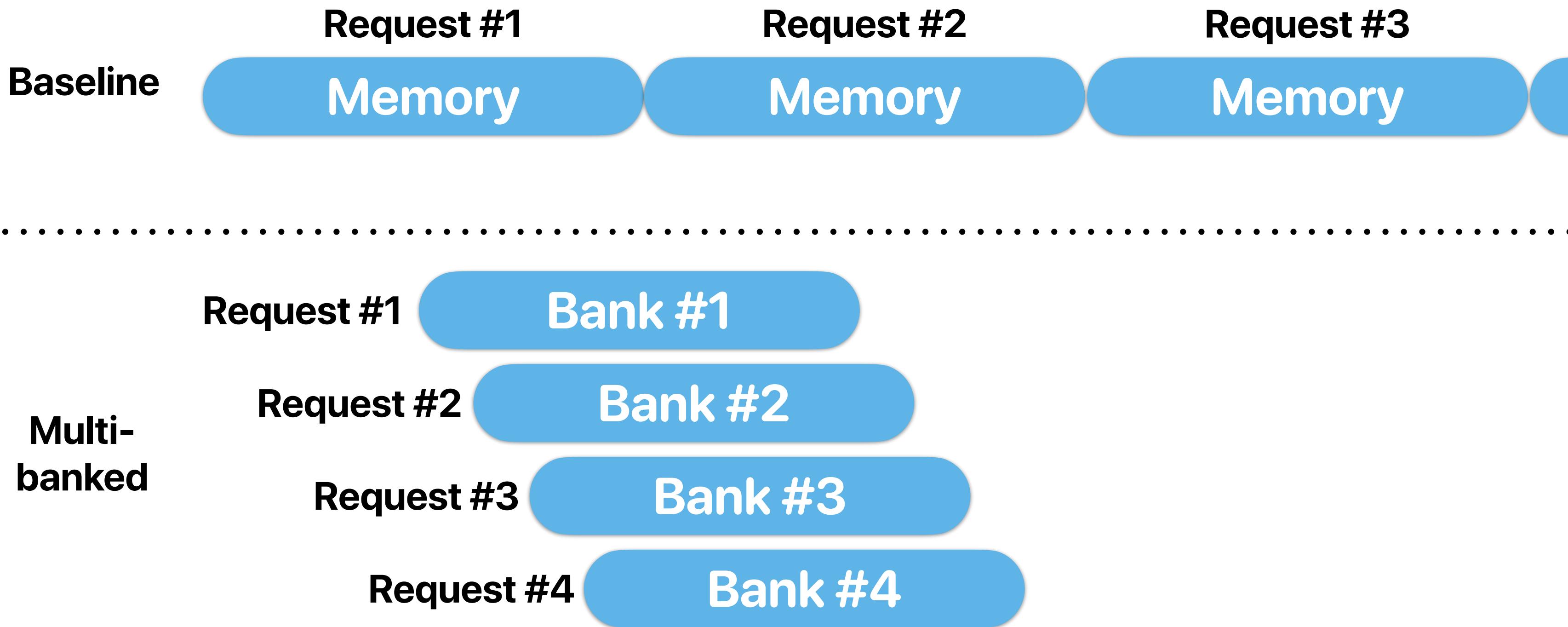
Blocking cache



Multibanks & non-blocking caches

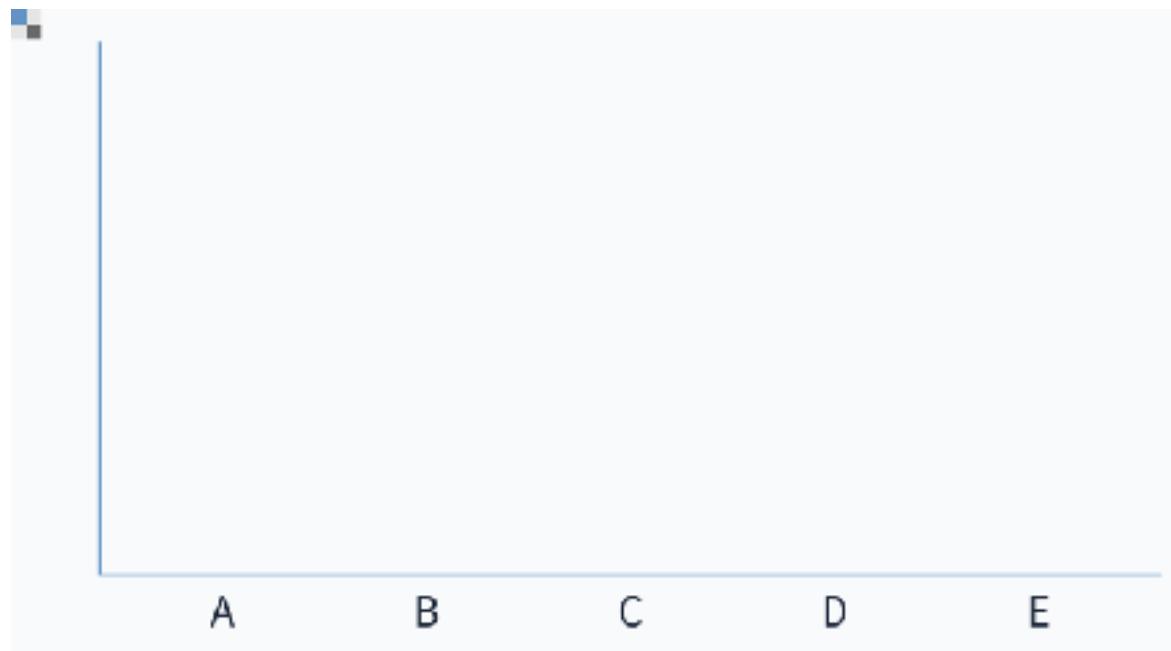


Pipelined access and multi-banked caches



Pipelined access and multi-banked caches

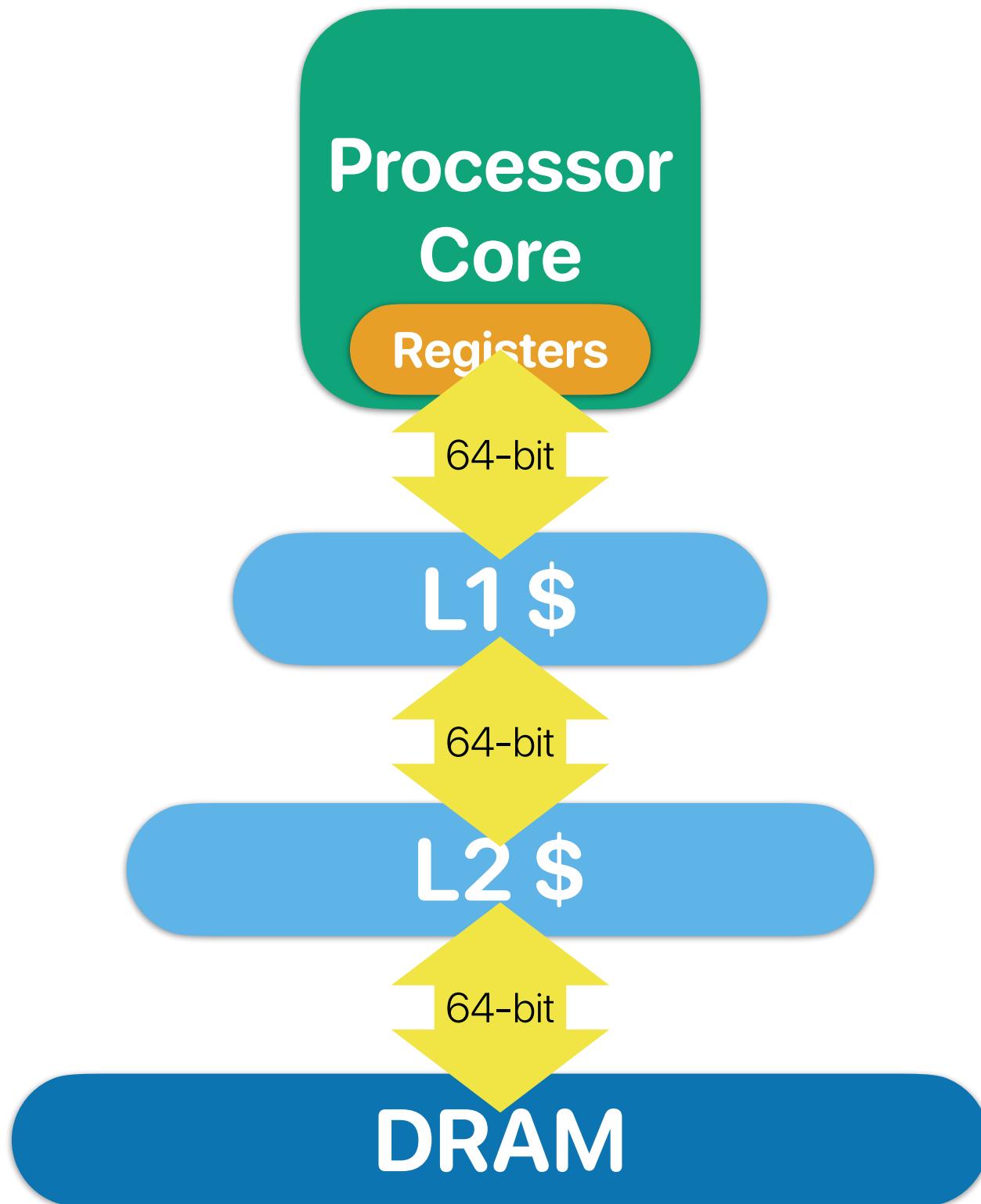
- Assume each bank in the \$ takes 10 ns to serve a request, and the \$ can take the next request 1 ns after assigning a request to a bank — if we have 4 banks and we want to serve 4 requests, what's the speedup over non-banked, non-pipelined \$? — pick the closest one
 - A. 1x — no speedup
 - B. 2x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



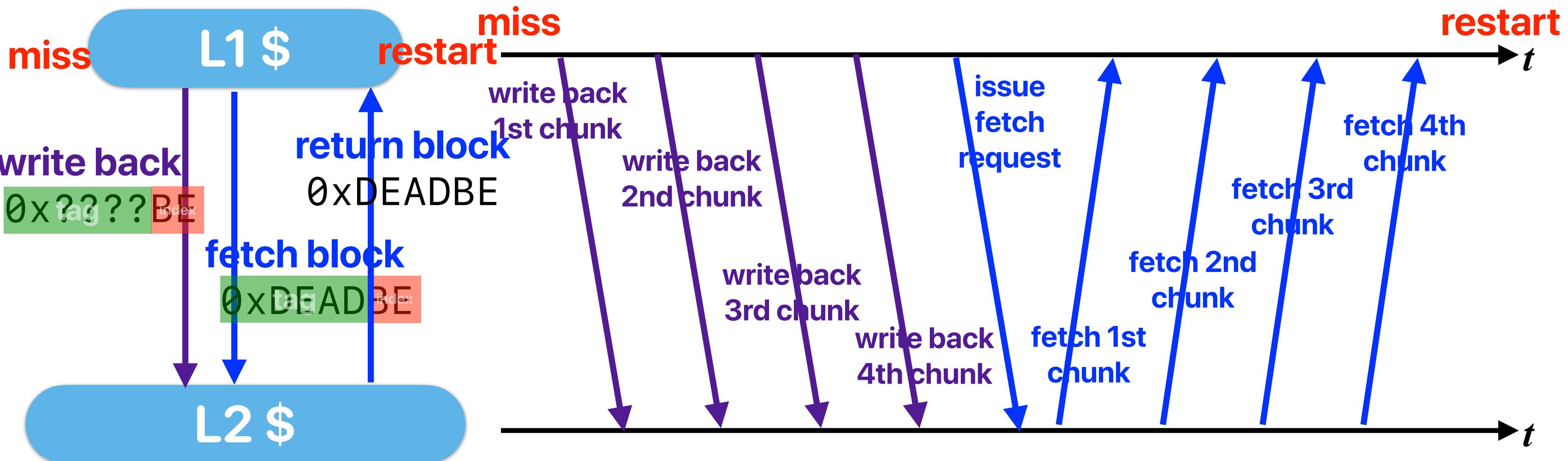
Pipelined access and multi-banked caches

- 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
- $$ET_{baseline} = 4 \times 10 \text{ ns} = 40 \text{ ns}$$
- $$ET_{banked} = 10 \text{ ns} + 3 \times 1 \text{ ns} = 13 \text{ ns}$$
- $$\begin{aligned} Speedup &= \frac{Execution\ Time_{baseline}}{Execution\ Time_{banked}} \\ &= \frac{40}{13} = 3.08 \times \end{aligned}$$

The bandwidth between units is limited

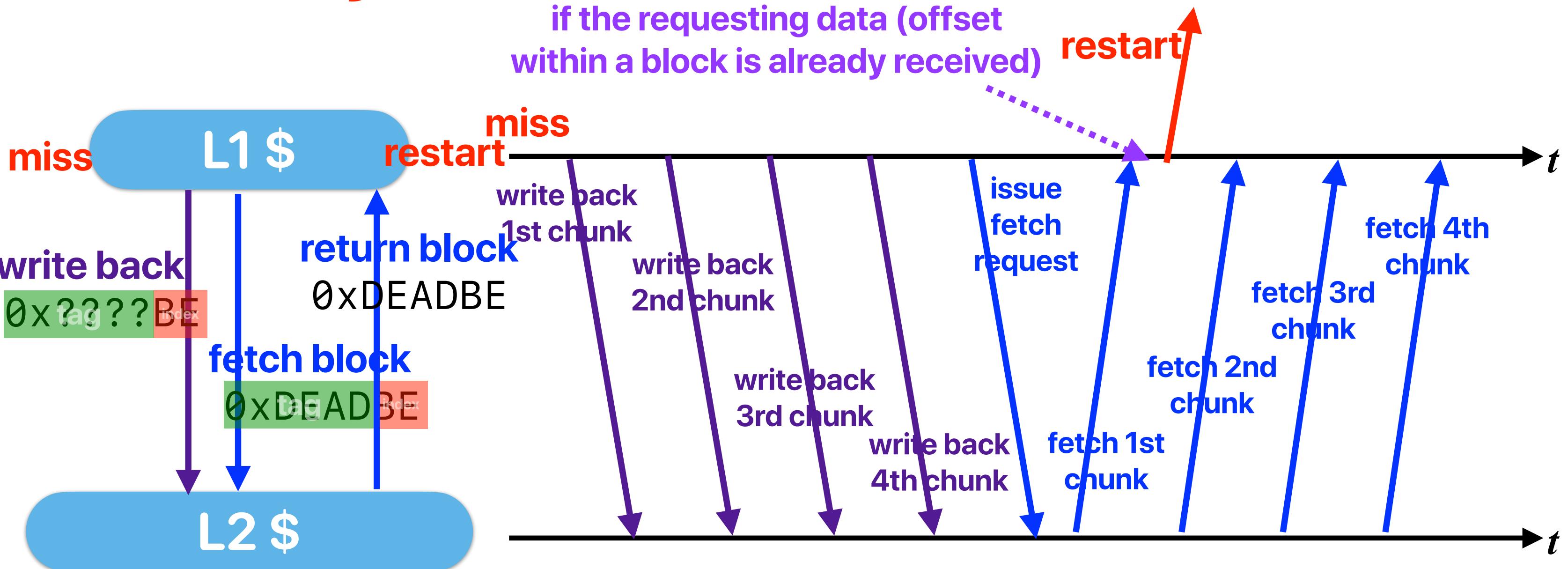


When we handle a miss



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

Early Restart and Critical Word First

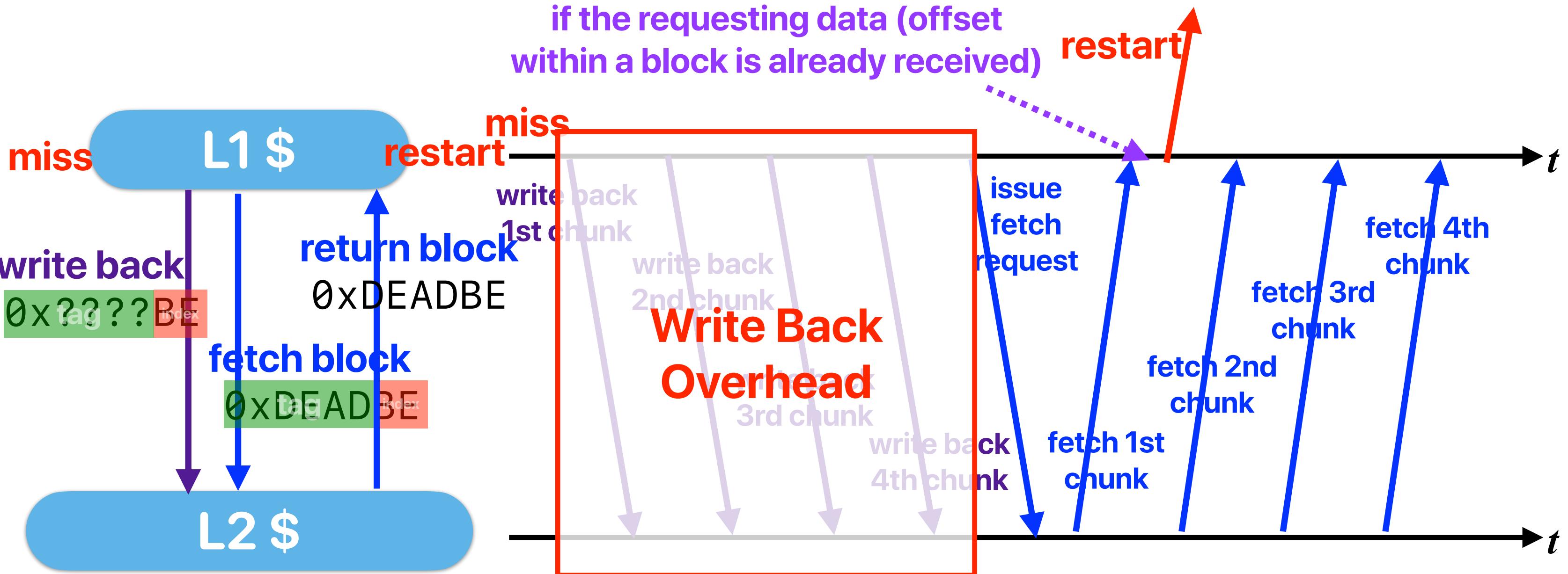


assume the bus between L1/L2 only allows a quarter of the cache block go through it

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).

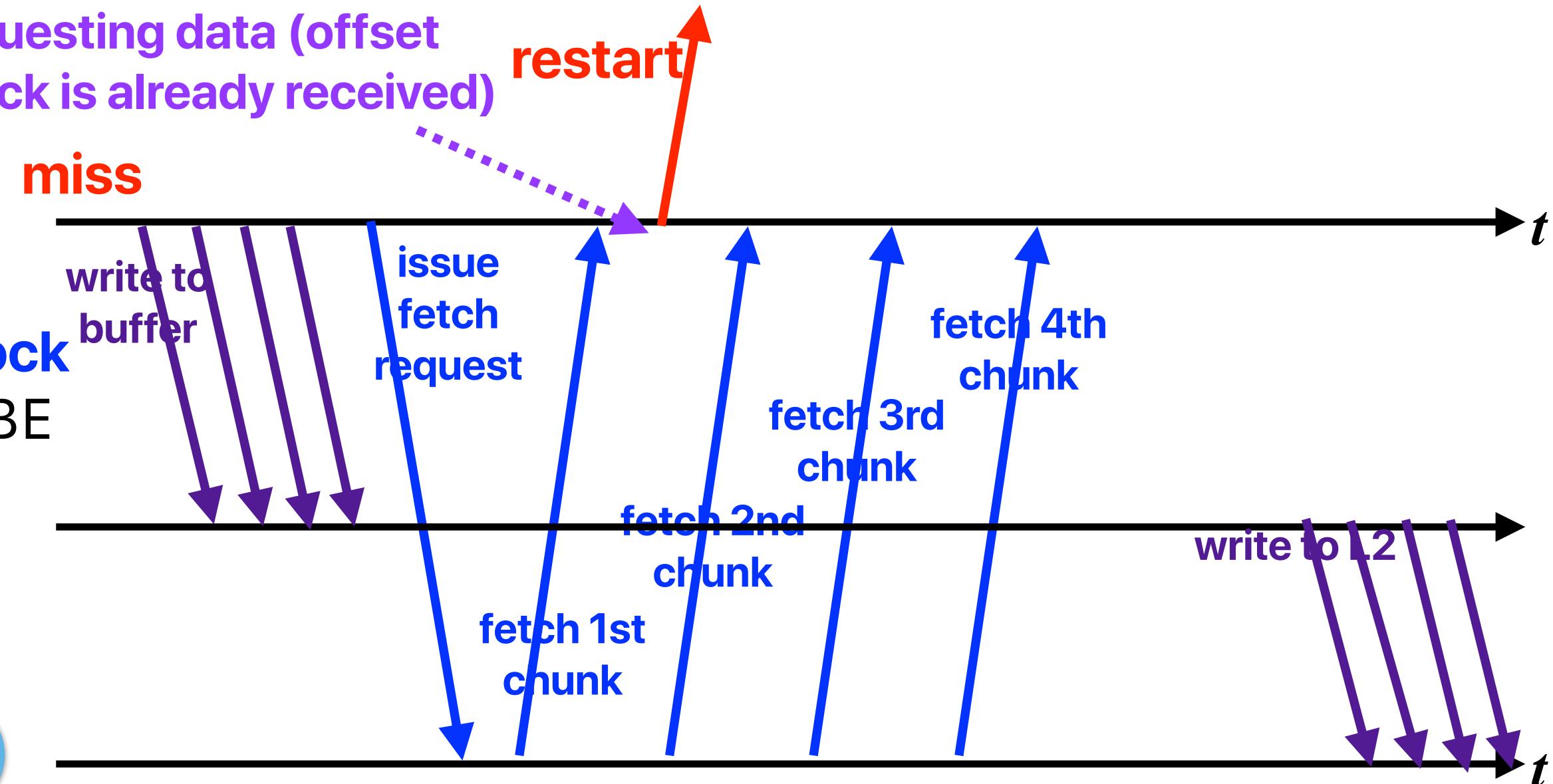
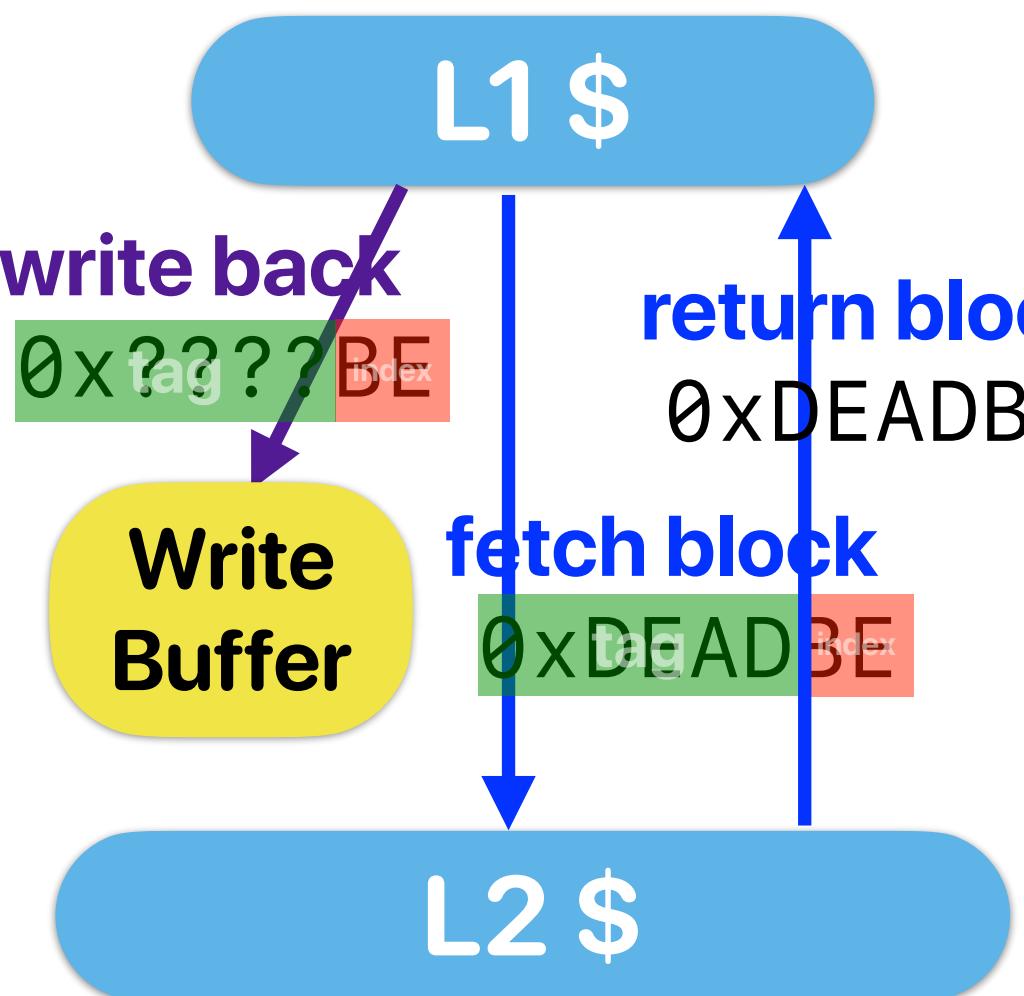
Can we avoid the overhead of writes?



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

Write buffer!

if the requesting data (offset within a block is already received)



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 Hardware Optimizations

- Hardware
 - Prefetch — compulsory miss
 - Stream buffer — compulsory miss
 - Miss cache — conflict miss
 - Victim cache — conflict miss
 - Write buffer — miss penalty
 - Bank/pipeline — miss penalty
 - Critical word first and early restart — miss panelty

How can programmer improve memory performance?

Data layout

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

Database table layout

- 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?

A

```
struct record
{
    int id;
    double *assignments;
    double average;
};

struct table
{
    struct record *records
};
```

B

```
struct table
{
    int *id;
    double **assignments;
    double *average;
};
```

Array of structures or structure of arrays

	Array of objects	object of arrays
	<pre>struct grades { int id; double *homework; double average; };</pre>	<pre>struct grades { int *id; double **homework; double *average; };</pre>
average of each homework	<pre>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; }</pre>	<pre>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; }</pre>

What data structure is performing better

	Array of objects	object of arrays
	<pre>struct grades { int id; double *homework; double average; };</pre>	<pre>struct grades { int *id; double **homework; double *average; };</pre>
average of each homework	<pre>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; }</pre>	<pre>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; }</pre>

Column-store or row-store

- If you're designing an in-memory database system, will you be using

RowId	Empld	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, Firsntname 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];  
    }  
}
```

$O(n^2)$

Same

Same

Better

Complexity

Instruction Count?

Clock Rate

CPI

$O(n^2)$

Same

Same

Worse

Loop interchange

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%

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[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]; //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%

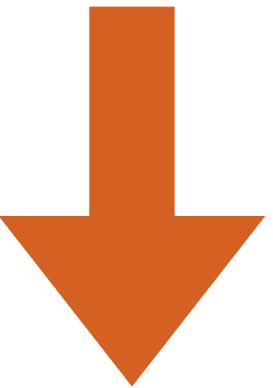
Loop fission

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 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];
}
```

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

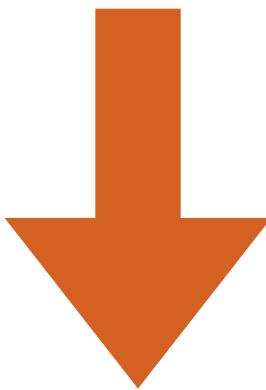
```
double a[8192], b[8192], c[8192], \
       d[8192], e[8192];
for(i = 0; i < 512; i++) {
    e[i] = (a[i] * b[i] + c[i])/d[i];
}
```

Loop optimizations

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];
}
```

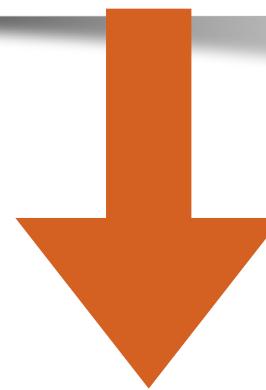
Loop fission



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];
```

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];
}
```

Blocking

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 $O(n^3)$

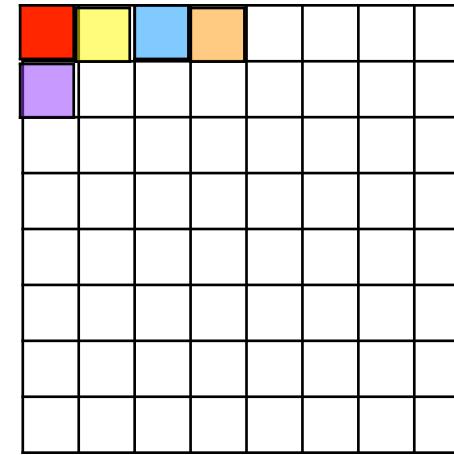
If $n=1024$, it takes about 1 sec

How long is it take when $n=2048$?

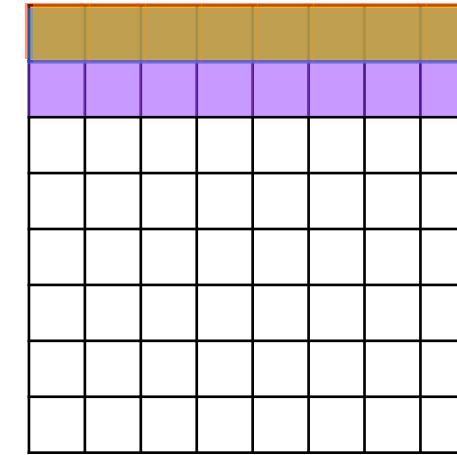
Matrix Multiplication

```
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];  
        }  
    }  
}
```

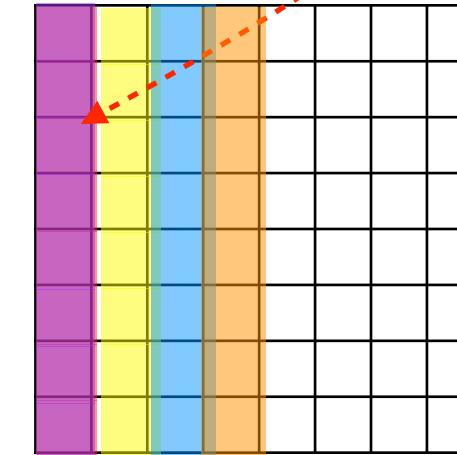
Very likely a miss if
array is large



c



a

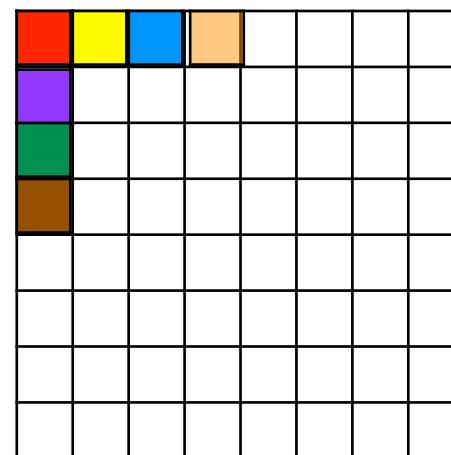
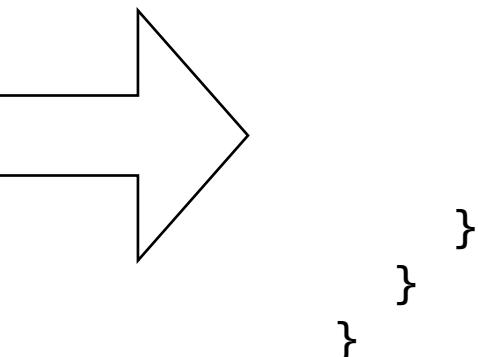


b

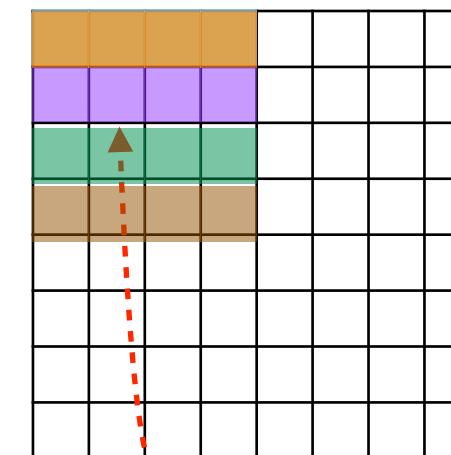
- If each dimension of your matrix is 2048
 - Each row takes 2048×8 bytes = 16KB
 - The L1 \$ of intel Core i7 is 32KB, 8-way, 64-byte blocked
 - You can only hold at most 2 rows/columns of each matrix!
 - You need the same row when j increase!

Block algorithm for matrix multiplication

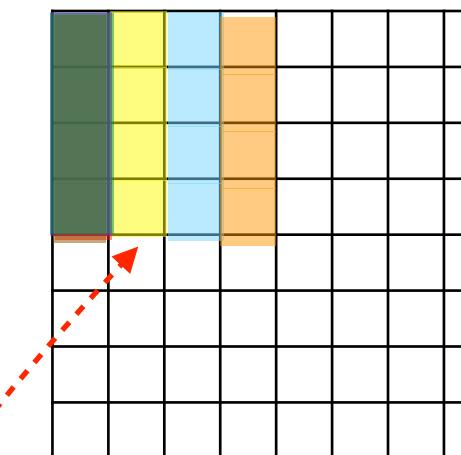
```
for(i = 0; i < ARRAY_SIZE; i++) {  
    for(j = 0; j < ARRAY_SIZE; j++) {  
        for(k = 0; k < ARRAY_SIZE; k++) {  
            c[i][j] += a[i][k]*b[k][j];  
        }  
    }  
}
```



c



a



b

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

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

- Comparing the naive algorithm and block algorithm on matrix multiplication, what kind of misses does block 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

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

- Comparing the naive algorithm and block algorithm on matrix multiplication, what kind of misses does block 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

Summary of Software Optimizations

- Software
 - Data layout — capacity miss (check assignment), conflict miss, compulsory miss
 - Blocking — 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

Announcement

- Regarding assignments 2
 - Due this Friday
 - The programming assignment is NOT EASY/NOT TRIVIAL and I only see one turned in
 - 1652 jobs has been sent to our servers — each of you submit only 24 jobs on average
- Regarding midterm
 - Midterm will be on gradescope
 - You will have a 80-minute slot when you pick within a 24-hour timeframe of 10/30/2022 12:00pm — 10/31/2022 12:00pm
 - Once you start, you cannot stop until you finish.
 - Only one shot
 - You cannot ask ANY question on piazza or ANYONE during this timeframe
 - You cannot expect the TA/instructor to respond to your e-mail during your test
 - You will be able to use LaTeX syntax and you should
 - We will release a midterm review next Wednesday

Computer Science & Engineering

203

つづく

