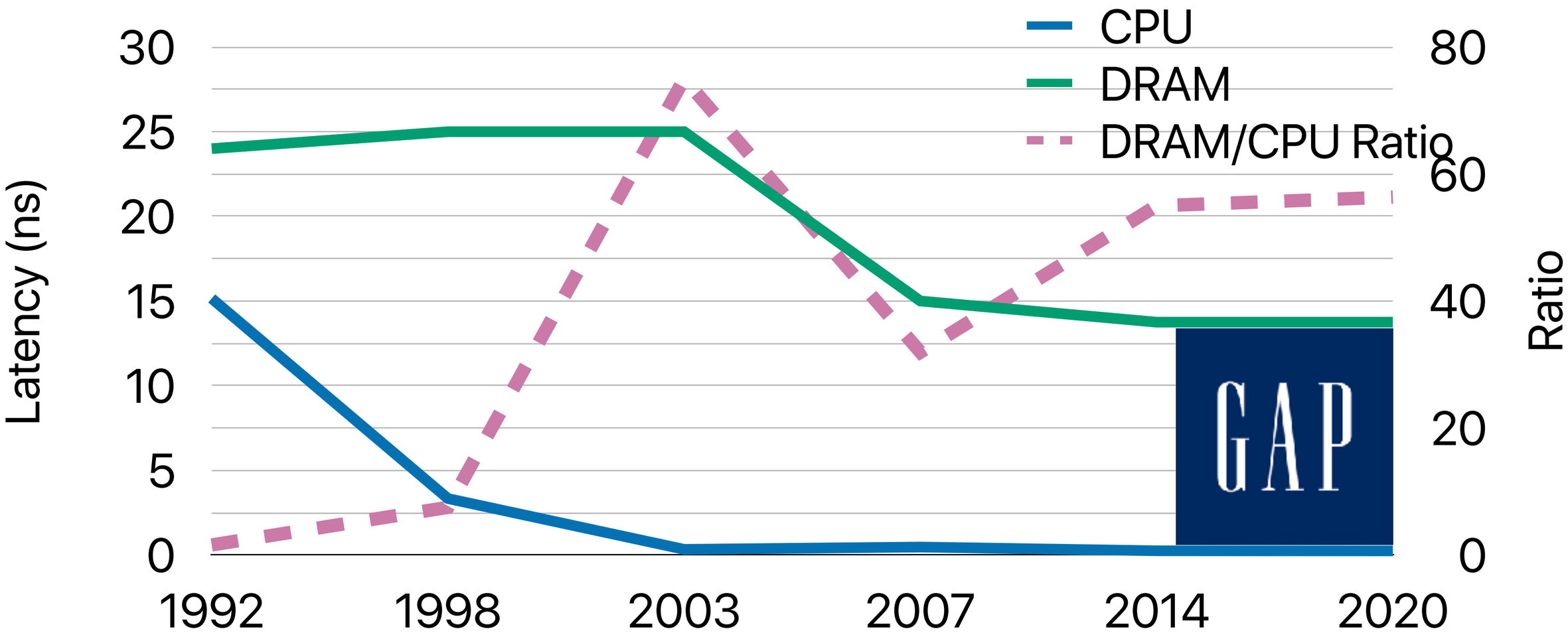
# Memory Hierarchy (4): Cache Misses and How to Address Them — the Software Version

Hung-Wei Tseng

# Recap: The “latency” gap between CPU and DRAM




| CPU | i486 | Pentium II | Pentium 4 | Core 2 | Core i7-4790K | Core i5-10600K |
|---|---|---|---|---|---|---|
| DRAM | SDRAM | DDR | DDR2 | DDR3 | DDR4 | DDR5 |

# Recap: Memory Hierarchy

fastest

< 1ns

a few ns

tens of ns

us/ms

Processor

Processor Core

Registers

SRAM $

DRAM

Storage

TBs

larger

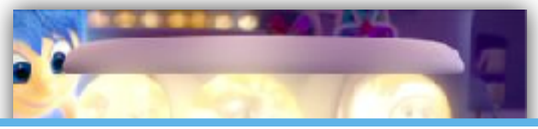
fastest

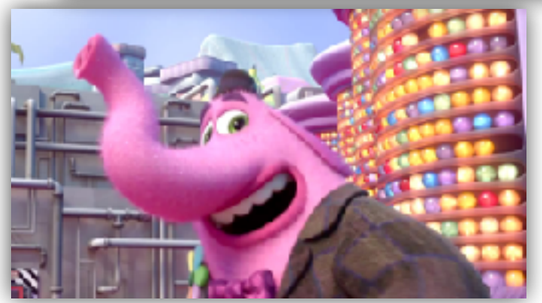
L1 $

L2 $

L3 $

larger





# Recap: 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 < 8192; 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 | Miss | |
| b[0] | 0x20000 | 0b0010000000000000000000 | 0x10 | 0x0 | Miss | |
| c[0] | 0x30000 | 0b0011000000000000000000 | 0x18 | 0x0 | Miss | |
| d[0] | 0x40000 | 0b0100000000000000000000 | 0x20 | 0x0 | Miss | |
| e[0] | 0x50000 | 0b0101000000000000000000 | 0x28 | 0x0 | Miss | a[0–7] |
| a[1] | 0x10008 | 0b0001000000000000001000 | 0x8 | 0x0 | Miss | b[0–7] |
| b[1] | 0x20008 | 0b0010000000000000001000 | 0x10 | 0x0 | Miss | c[0–7] |
| c[1] | 0x30008 | 0b0011000000000000001000 | 0x18 | 0x0 | Miss | d[0–7] |
| d[1] | 0x40008 | 0b0100000000000000001000 | 0x20 | 0x0 | Miss | e[0–7] |
| e[1] | 0x50008 | 0b0101000000000000001000 | 0x28 | 0x0 | Miss | a[0–7] |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

# Recap: NVIDIA Tegra X1 (cont.)

- 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 < 8192; 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]
```

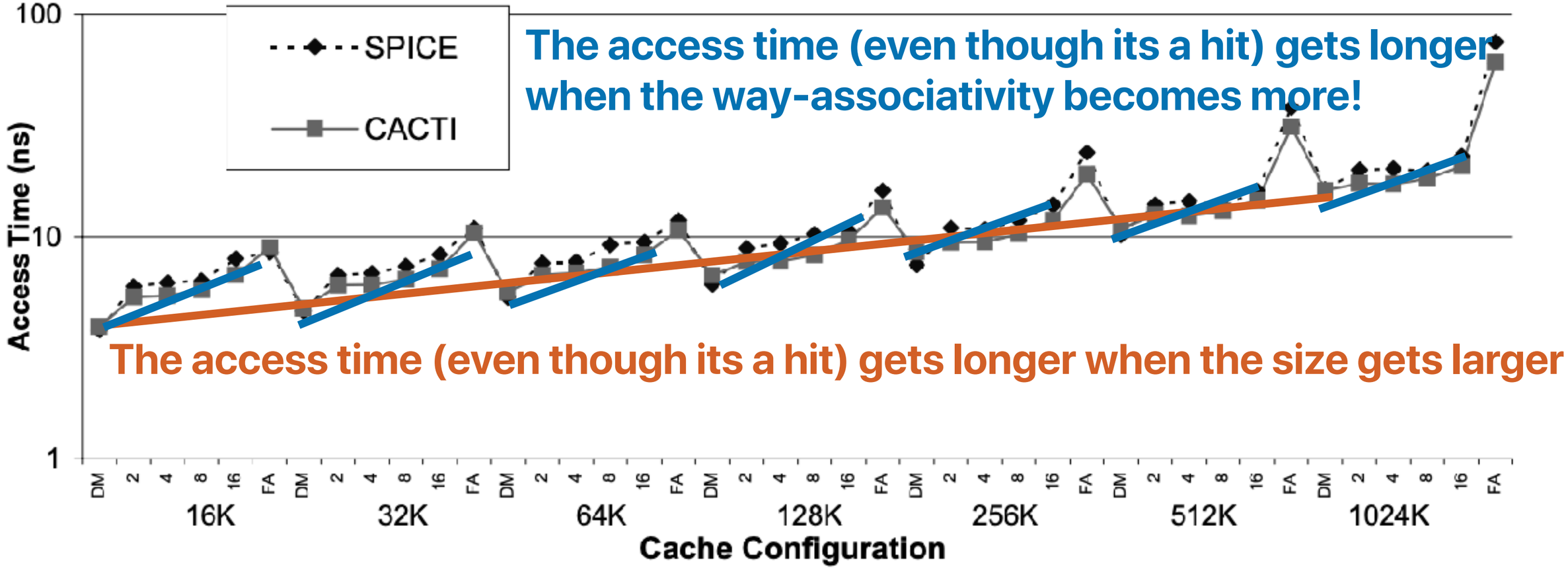
C = ABS
32KB = 4 * 64 * S
S = 128
offset = lg(64) = 6 bits
index = lg(128) = 7 bits
tag = the rest bits

tag index offset

| | Address (Hex) | Address in binary | Tag | Index | Hit? Miss? | Replace? |
|---|---|---|---|---|---|---|
| a[7] | 0x10038 | 0b0001000000000111000 | 0x8 | 0x0 | Miss | |
| b[7] | 0x20038 | 0b0010000000000111000 | 0x10 | 0x0 | Miss | |
| c[7] | 0x30038 | 0b0011000000000111000 | 0x18 | 0x0 | Miss | |
| d[7] | 0x40038 | 0b0100000000000111000 | 0x20 | 0x0 | Miss | |
| e[7] | 0x50038 | 0b0101000000000111000 | 0x28 | 0x0 | Miss | a[0–7] |
| a[8] | 0x10040 | 0b0001000000001000000 | 0x8 | 0x1 | Miss | |
| b[8] | 0x20040 | 0b0010000000001000000 | 0x10 | 0x1 | Miss | |
| c[8] | 0x30040 | 0b0011000000001000000 | 0x18 | 0x1 | Miss | |
| d[8] | 0x40040 | 0b0100000000001000000 | 0x20 | 0x1 | Miss | |
| e[8] | 0x50040 | 0b0101000000001000000 | 0x28 | 0x1 | Miss | a[8–15] |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

**100% miss rate!**

# Recap: Cache configurations and accessing time




Glen Reinman and Norman P. Jouppi. CACTI 2.0: An Integrated Cache Timing and Power Model. WRL Research Report 2000/7
https://bitsavers.trailing-edge.com/pdf/dec/tech_reports/WRL-2000-7.pdf

# Outline

- Architectural support for optimizing cache performance (cont.)
- Optimizing your code for better cache performance!

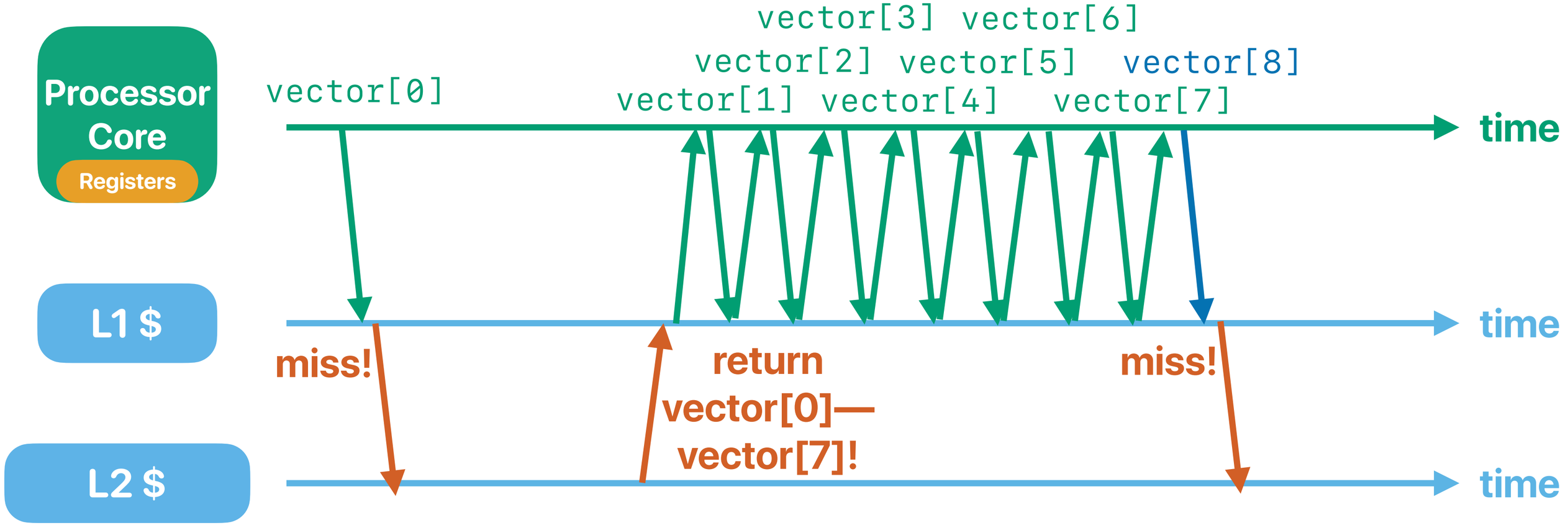
# Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers

**Norman P. Jouppi**

# Prefetching

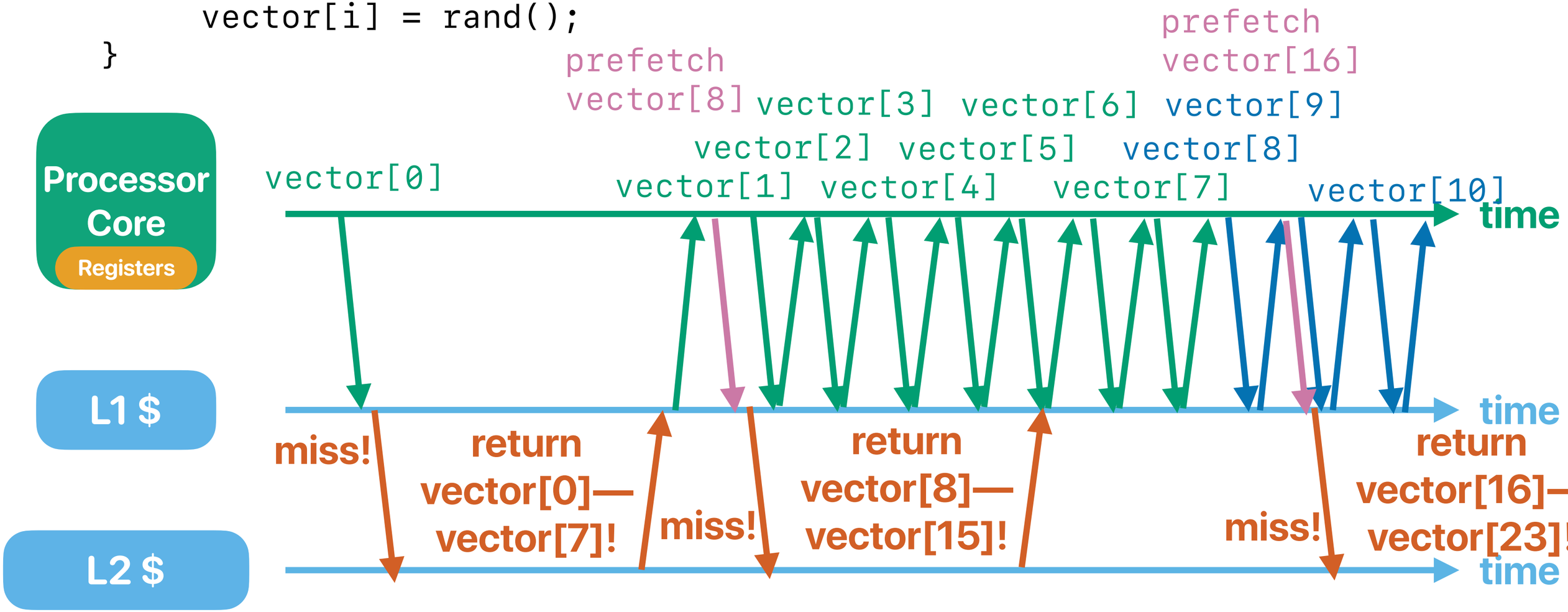
# Spatial locality revisited

```
for(i = 0;i < size; i++) {
      vector[i] = rand();
}
```



# What if we "pre-"fetch the next line?

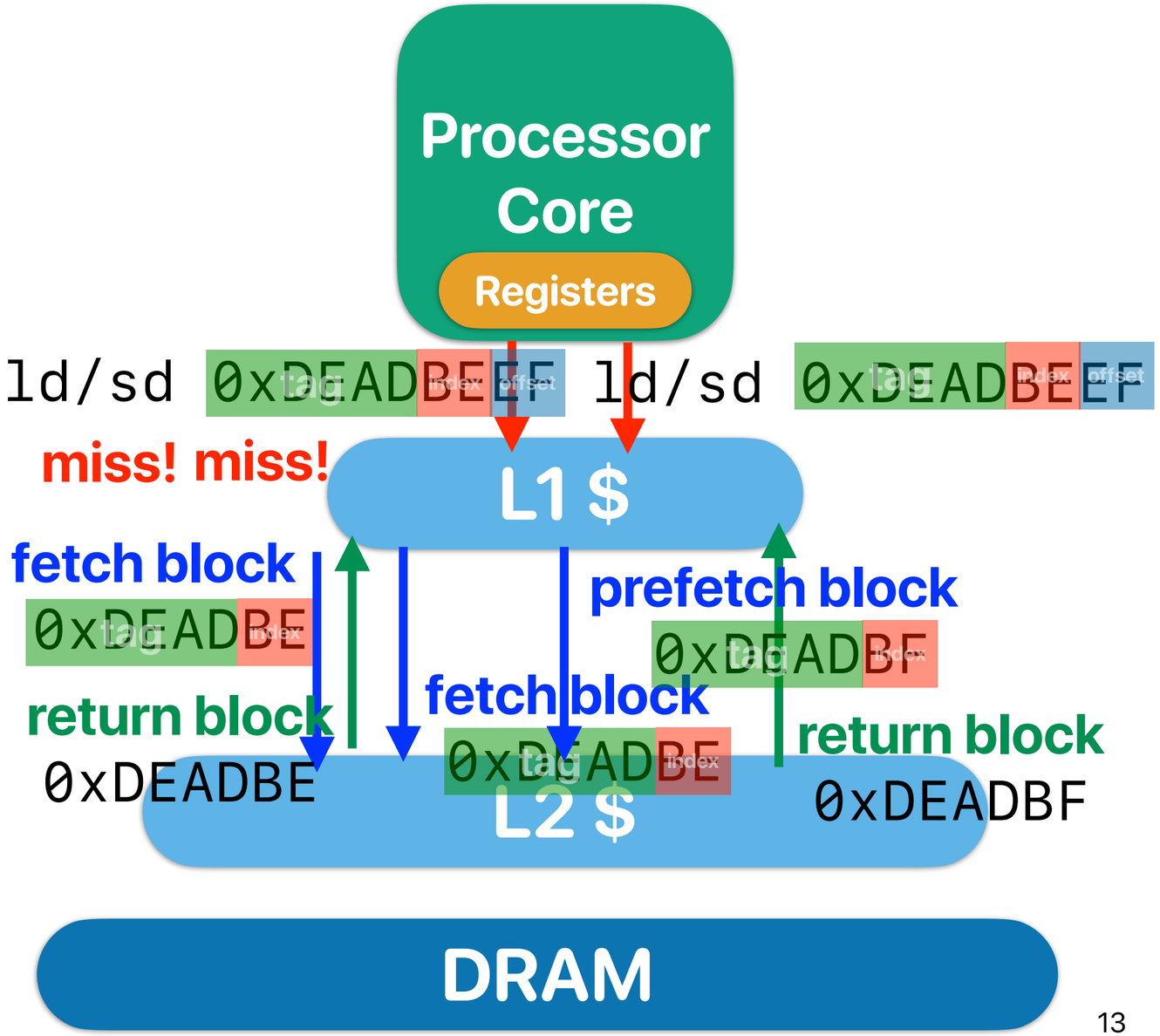
```
for(i = 0;i < size; i++) {
      vector[i] = rand();
}
```



# Hardware Prefetching

- The hardware 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
- The processor can keep track the distance between misses. If there is a pattern between misses, fetch `miss_data_address + offset` for a miss

# 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

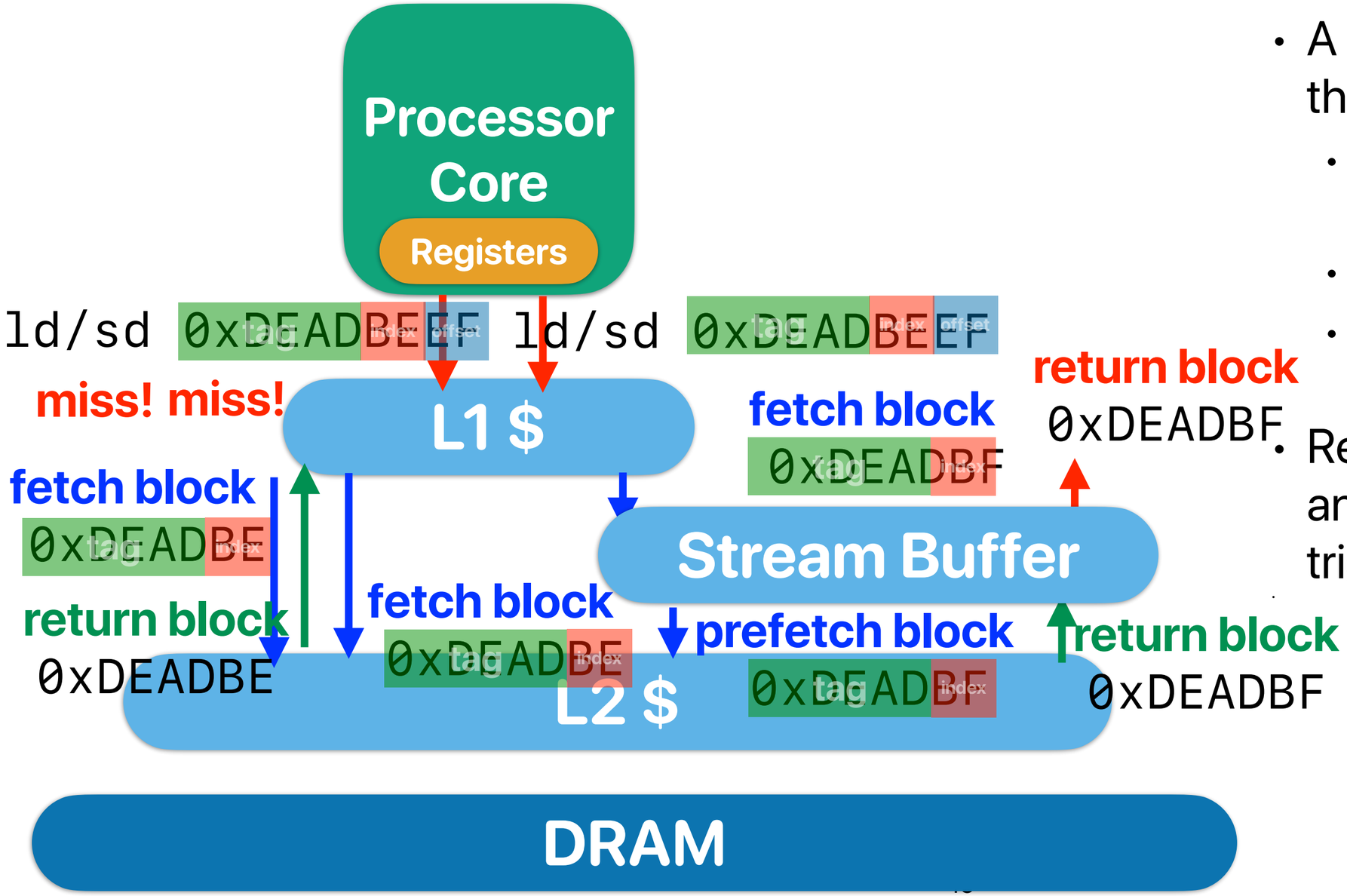
tag index offset

| | Address (Hex) | Address in binary | Tag | Index | Hit? Miss? | Replace? | Prefetch |
|---|---|---|---|---|---|---|---|
| a[0] | 0x10000 | 0b0001000000000000000000 | 0x8 | 0x0 | Miss | | a[8–15] |
| b[0] | 0x20000 | 0b0010000000000000000000 | 0x10 | 0x0 | Miss | | b[8–15] |
| c[0] | 0x30000 | 0b0011000000000000000000 | 0x18 | 0x0 | Miss | | c[8–15] |
| d[0] | 0x40000 | 0b0100000000000000000000 | 0x20 | 0x0 | Miss | | d[8–15] |
| e[0] | 0x50000 | 0b0101000000000000000000 | 0x28 | 0x0 | Miss | a[0–7] | e[8–15] |
| a[1] | 0x10008 | 0b0001000000000000001000 | 0x8 | 0x0 | Miss | b[0–7] | |
| b[1] | 0x20008 | 0b0010000000000000001000 | 0x10 | 0x0 | Miss | c[0–7] | |
| c[1] | 0x30008 | 0b0011000000000000001000 | 0x18 | 0x0 | Miss | d[0–7] | |
| d[1] | 0x40008 | 0b0100000000000000001000 | 0x20 | 0x0 | Miss | e[0–7] | |
| e[1] | 0x50008 | 0b0101000000000000001000 | 0x28 | 0x0 | Miss | a[0–7] | |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

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

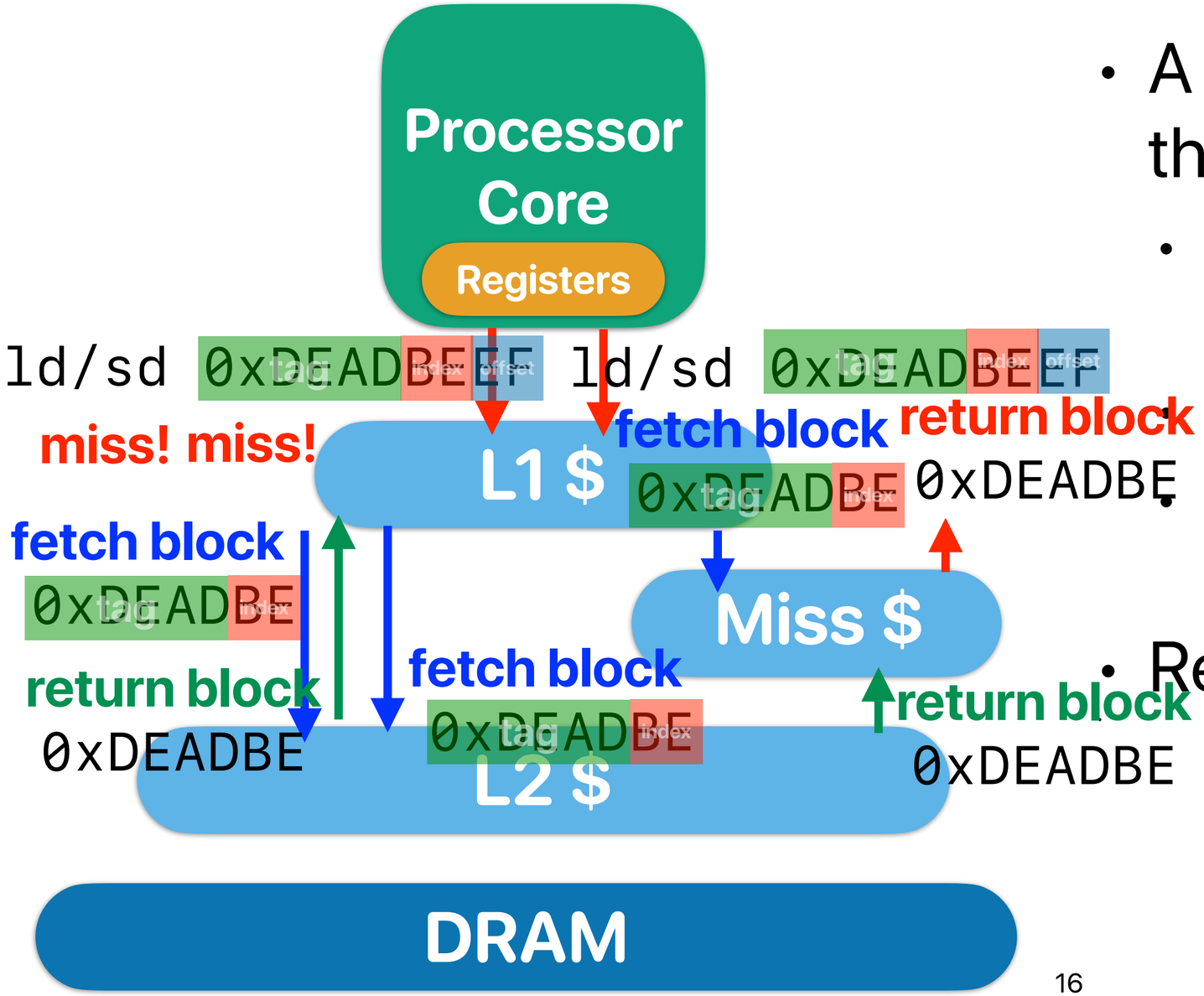
100% miss rate!

# 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

# NVIDIA Tegra X1 with miss caching

- 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]
```

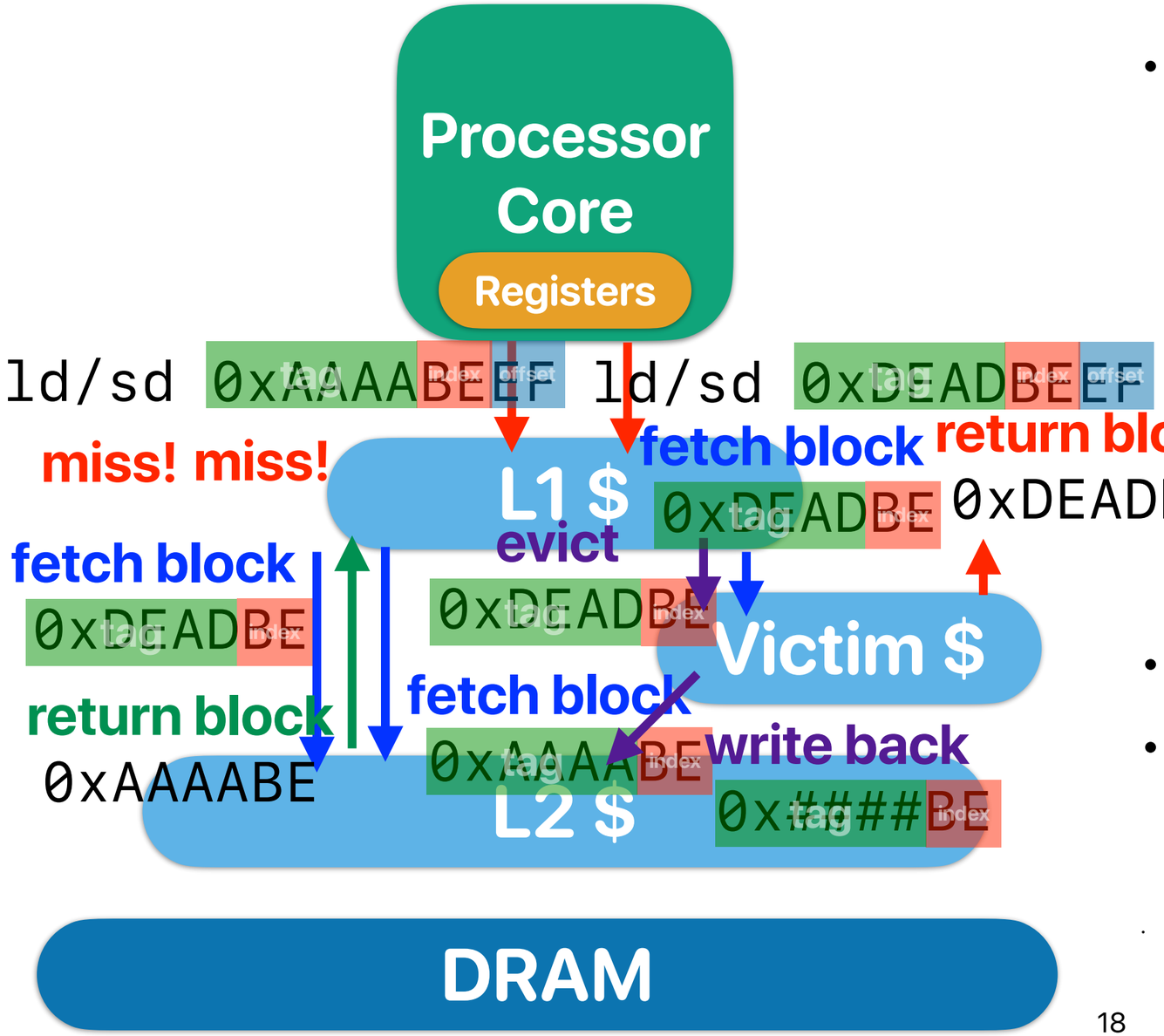
C = ABS
32KB = 4 * 64 * S
S = 128
offset = lg(64) = 6 bits
index = lg(128) = 7 bits
tag = the rest bits

tag index offset

| | Address (Hex) | Address in binary | Tag | Index | Hit? Miss? | Replace? | Miss caching |
|---|---|---|---|---|---|---|---|
| a[0] | 0x10000 | 0b0001000000000000000000 | 0x8 | 0x0 | Miss | | a[0–7] |
| b[0] | 0x20000 | 0b0010000000000000000000 | 0x10 | 0x0 | Miss | | a,b[0–7] |
| c[0] | 0x30000 | 0b0011000000000000000000 | 0x18 | 0x0 | Miss | | a,b,c[0–7] |
| d[0] | 0x40000 | 0b0100000000000000000000 | 0x20 | 0x0 | Miss | | a,b,c,d[0–7] |
| e[0] | 0x50000 | 0b0101000000000000000000 | 0x28 | 0x0 | Miss | a[0–7] | a,b,c,d,e[0–7] |
| a[1] | 0x10008 | 0b0001000000000000001000 | 0x8 | 0x0 | Hit in miss $ | b[0–7] | |
| b[1] | 0x20008 | 0b0010000000000000001000 | 0x10 | 0x0 | Hit in miss $ | c[0–7] | |
| c[1] | 0x30008 | 0b0011000000000000001000 | 0x18 | 0x0 | Hit in miss $ | d[0–7] | |
| d[1] | 0x40008 | 0b0100000000000000001000 | 0x20 | 0x0 | Hit in miss $ | e[0–7] | |
| e[1] | 0x50008 | 0b0101000000000000001000 | 0x28 | 0x0 | Hit in miss $ | a[0–7] | |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | |

Need 5 entries

# 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

# NVIDIA Tegra X1 with victim cache

- 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? | Victim $ |
|---|---|---|---|---|---|---|---|
| a[0] | 0x10000 | 0b0001000000000000000000 | 0x8 | 0x0 | Miss | | |
| b[0] | 0x20000 | 0b0010000000000000000000 | 0x10 | 0x0 | Miss | | |
| c[0] | 0x30000 | 0b0011000000000000000000 | 0x18 | 0x0 | Miss | | |
| d[0] | 0x40000 | 0b0100000000000000000000 | 0x20 | 0x0 | Miss | | |
| e[0] | 0x50000 | 0b0101000000000000000000 | 0x28 | 0x0 | Miss | a[0–7] | a[0–7] |
| a[1] | 0x10008 | 0b0001000000000000001000 | 0x8 | 0x0 | Hit in victim $ | b[0–7] | b[0–7] |
| b[1] | 0x20008 | 0b0010000000000000001000 | 0x10 | 0x0 | Hit in victim $ | c[0–7] | c[0–7] |
| c[1] | 0x30008 | 0b0011000000000000001000 | 0x18 | 0x0 | Hit in victim $ | d[0–7] | d[0–7] |
| d[1] | 0x40008 | 0b0100000000000000001000 | 0x20 | 0x0 | Hit in victim $ | e[0–7] | e[0–7] |
| e[1] | 0x50008 | 0b0101000000000000001000 | 0x28 | 0x0 | Hit in victim $ | a[0–7] | a[0–7] |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

**Only need 1 entry!**

# 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

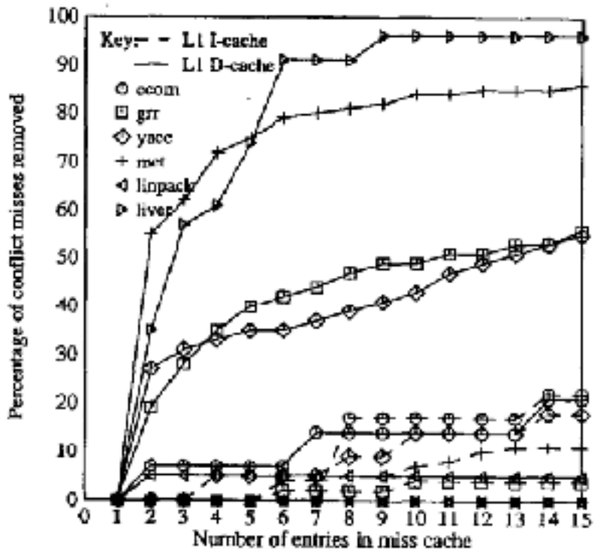



**Figure 3-3:** Conflict misses removed by miss caching

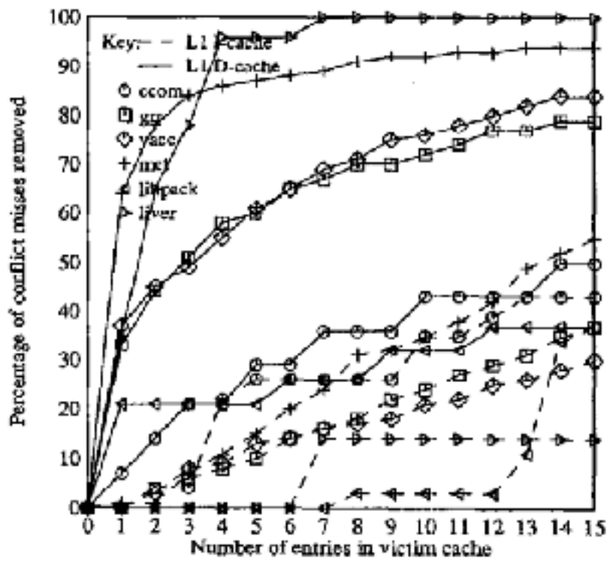



**Figure 3-5:** Conflict misses removed by victim caching

# 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

# Takeaways: Optimizing cache performance through hardware

- 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

# Takeaways: Optimizing cache performance through hardware

- 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

**We still need additional search time and power — still not ideal!**

# How can programmer improve memory performance?

# Data structures

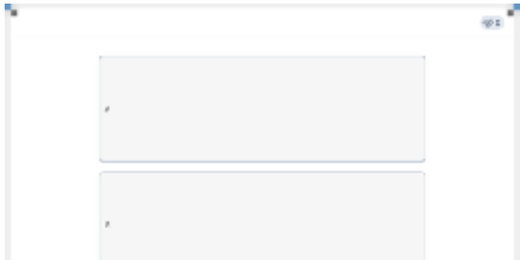
https://www.pollev.com/hungweitseng close in 1:30

# 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 {`<br>`  int id;`<br>`  double assignment_1, assignment_2, assignment 3, …;`<br>`};`<br><br>`table = (struct grades *) \`<br>`malloc(num_of_students*sizeof(struct grades));` | `struct grades {`<br>`  int *id;`<br>`  double *assignment_1, *assignment_2, *assignment_3, …;`<br>`};`<br><br>`table = (struct grades *)malloc(sizeof(struct grades));`<br>`table->assignment_1 = \`<br>`(double *)malloc(num_of_students*sizeof(double);`<br>`table->assignment_2 = \`<br>`(double *)malloc(num_of_students*sizeof(double);` |

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**

```
struct grades {
  int id;
  double assignment_1, assignment_2, assignment 3, …;
};

table = (struct grades *) \
malloc(num_of_students*sizeof(struct grades));
```

**object of arrays**

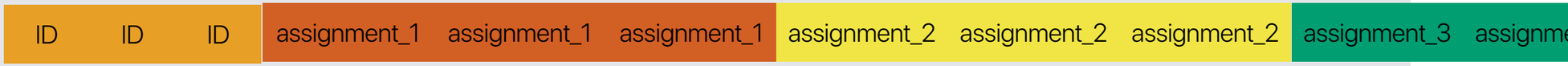
```
struct grades {
  int *id;
  double *assignment_1, *assignment_2, *assignment_3, …;
};

table = (struct grades *)malloc(sizeof(struct grades));
table->assignment_1 = \
(double *)malloc(num_of_students*sizeof(double);
table->assignment_2 = \
(double *)malloc(num_of_students*sizeof(double);
```

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 | object of arrays |
|---|---|---|
| | `struct grades { int id; double assignment_1, assignment_2, assignment 3, …; };` | `struct grades { int *id; double *assignment_1, *assignment_2, *assignment_3, …; };` |
| |  | |
| 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; }` |
| |  | |

# Column-store or row-store

- If you're designing an in-memory database system for the following table, 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;
```

# Take-aways: cache misses and their remedies

- 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
- Software optimizations
  - Data layout — change the order of storing data can improve capacity miss, conflict miss, compulsory miss

# 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];
  }
}
```

# Take-aways: cache misses and their remedies

- 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
- Software optimizations
  - Data layout — change the order of storing data can improve capacity miss, conflict miss, compulsory miss
  - Loop interchange — conflict/capacity misses

# Takeaways: Software Optimizations

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

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

**Only compulsory misses in e, a, b, c**

**Only compulsory misses in d**

**Capacity misses in e!**

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

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

**The working set of e is** $\frac{8192 \times 8B}{8B} = 8192\ blocks$

**The cache has** $\frac{32 \times 1024B}{64B} = 512 blocks$

$$miss\_rate = \frac{total\_\#\_misses}{total\_\#\_accesses} = \frac{\frac{8192}{8} \times 6}{8192 \times 4 + 8192 \times 2} = 0.125$$

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

# Takeaways: Software Optimizations

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

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

B

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

B

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

## Loop fusion

A

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

B

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

# Takeaways: Software Optimizations

- 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
- Software Optimization
  - 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

# 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

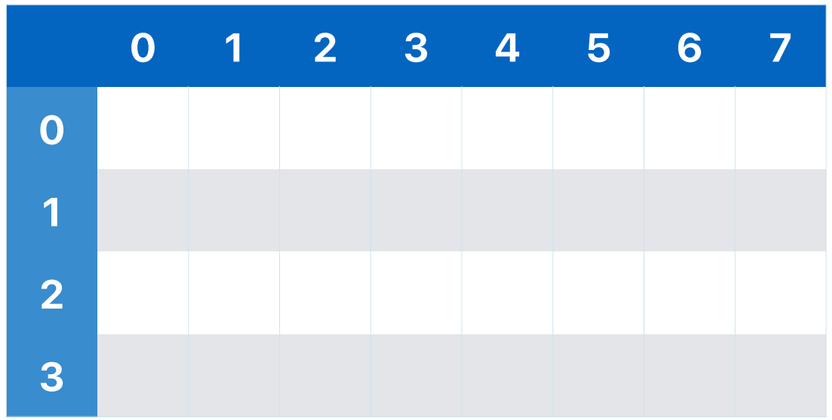
# Tiling/Blocking Algorithm

# What is an M by N "2-D" array in C?

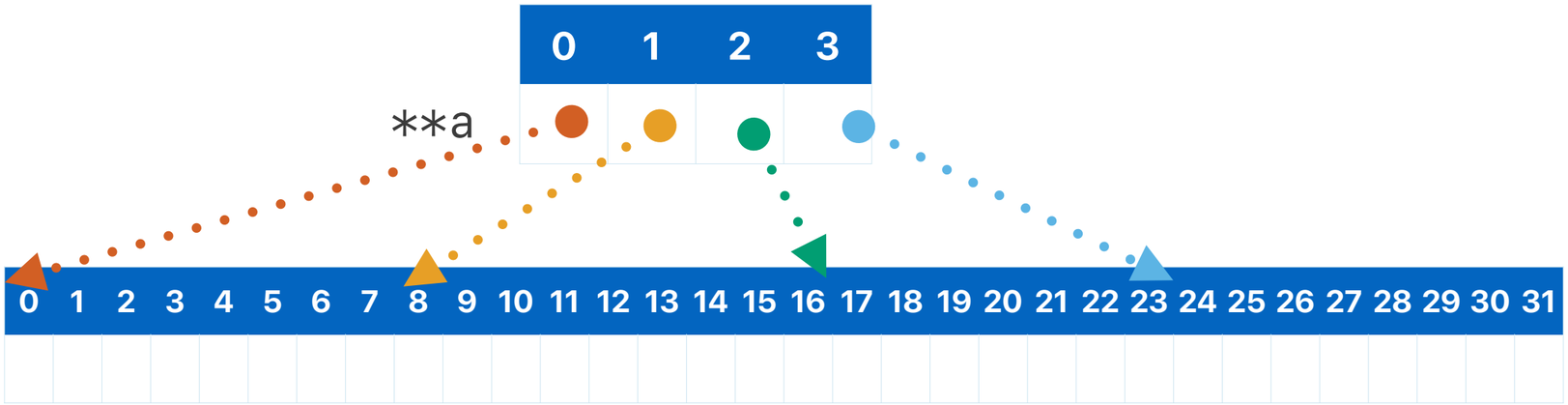
```
a = (double **)malloc(M*sizeof(double *));
for(i = 0; i < N; i++)
{
  a[i] = (double *)malloc(N*sizeof(double));
}
```

## a[i][j] is essentially a[i*N+j]

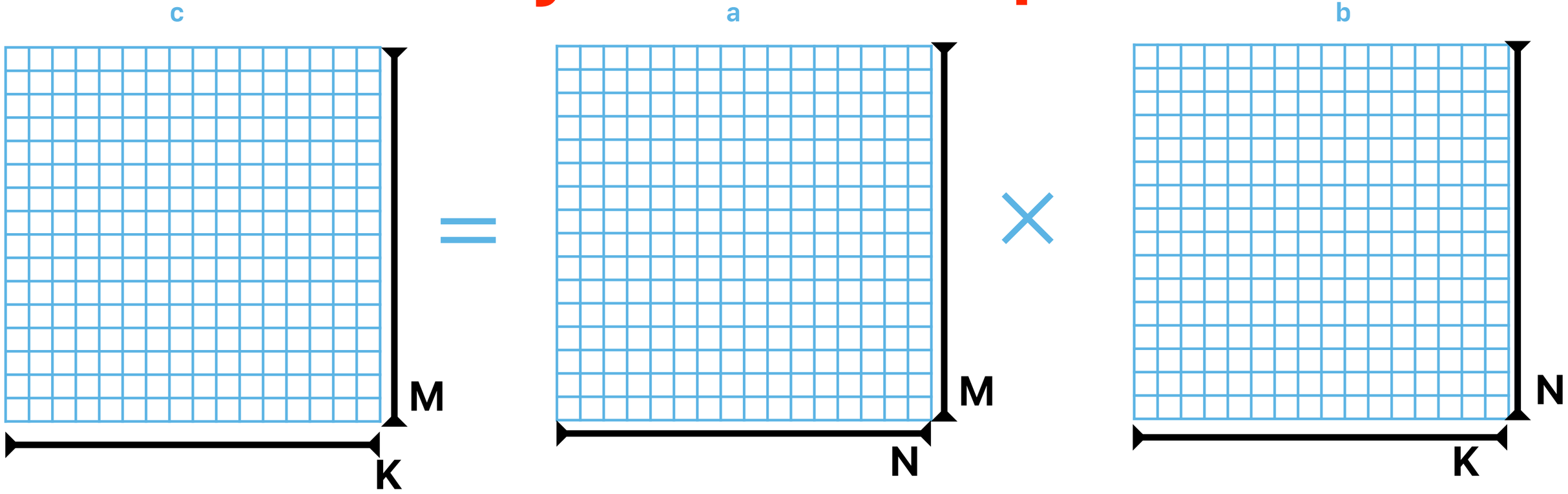
**abstraction**




**physical implementation**



# Case Study: Matrix Multiplications




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

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

**If M=N=K=1024, it takes about 2 sec**

**How long is it take when M=N=K=2048?**

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

# What kind(s) of misses are there in Matrix Multiplications

- Considering the case where M=N=K=2048, what do you think the majority type(s) of cache misses are we seeing on an intel processor with intel Core i7 is 48 KB, 12-way, 64-byte blocked L1-$?

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

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

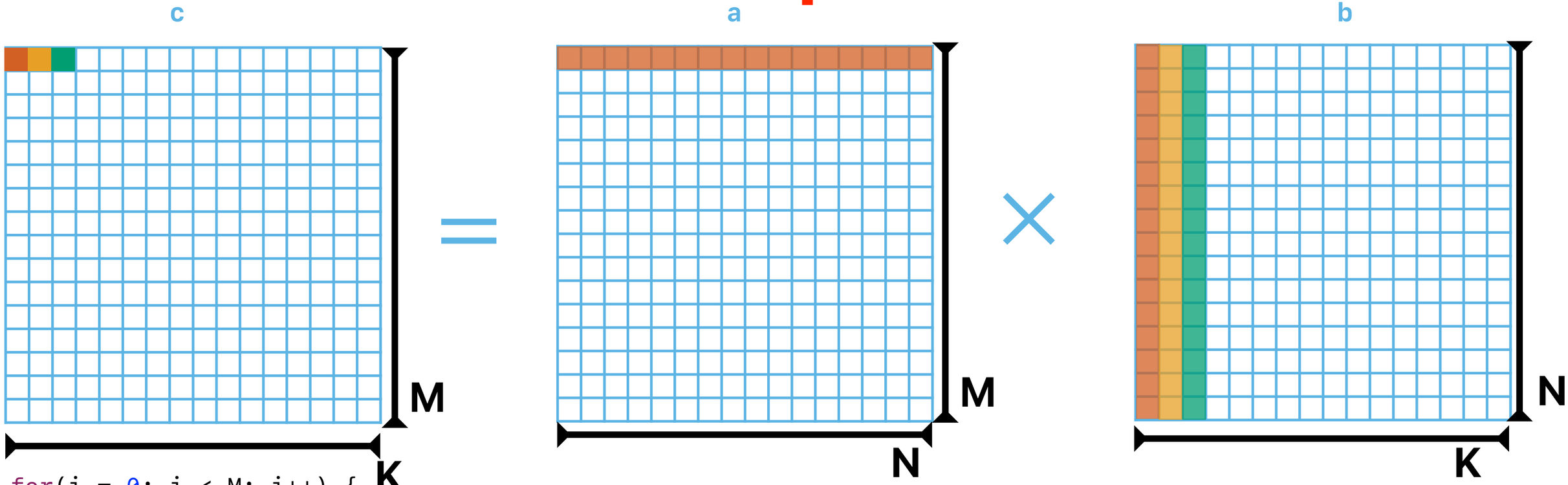
# What kind(s) of misses are there in Matrix Multiplications

- Considering the case where M=N=K=2048, what do you think the majority type(s) of cache misses are we seeing on an intel processor with intel Core i7 is 48 KB, 12-way, 64-byte blocked L1-$?

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

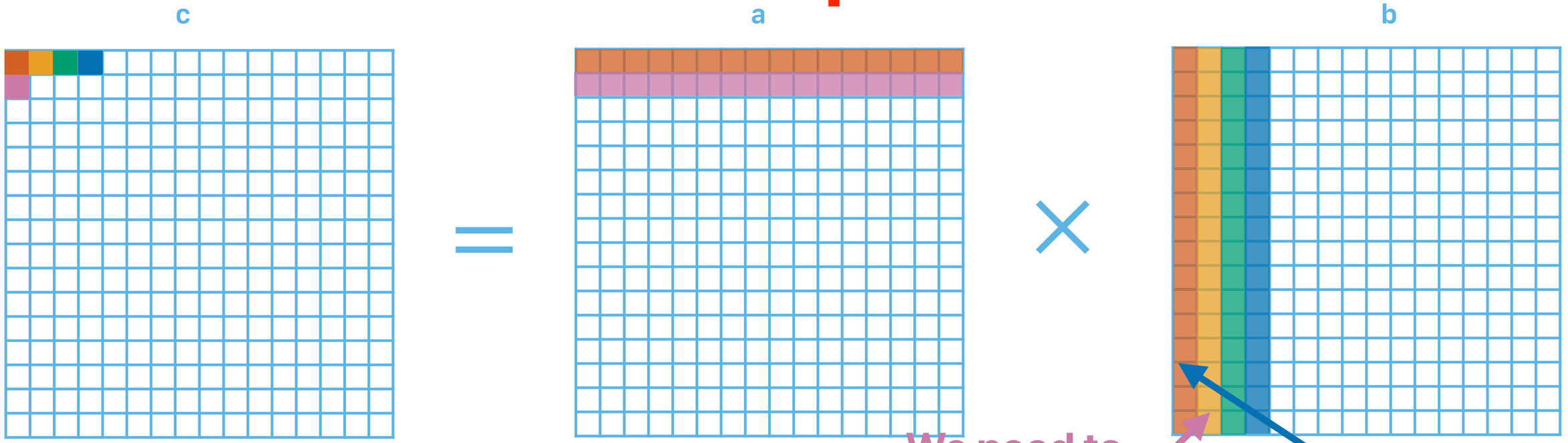
A. Compulsory miss
B. Capacity miss
C. Conflict miss
D. Capacity & conflict miss
E. Compulsory & conflict miss

# Matrix Multiplications




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

# Matrix Multiplications




- If each dimension of your matrix is 2048
  - Each row takes $2048 \times 8$ Bytes = 16 KB
  - Each column takes $\frac{2048 \times 8B}{8B} = 2048$ blocks
  - The L1-$ of intel Core i7 is 48 KB, 12-way, 64-byte blocked
  - You can only hold at most 3 rows or 0.25 of a column of each matrix!

We need to fetch everything again — capacity miss!

Unlikely to be kept in the cache

# What kind(s) of misses are there in Matrix Multiplications

- Considering the case where M=N=K=2048, what do you think the majority type(s) of cache misses are we seeing on an intel processor with intel Core i7 is 48 KB, 12-way, 64-byte blocked L1-$?

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

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

# Ideas regarding reducing misses in matrix multiplications

- Reducing capacity misses — we need to reduce the length of a row that we visit within a period of time

# Mathematical view of MM

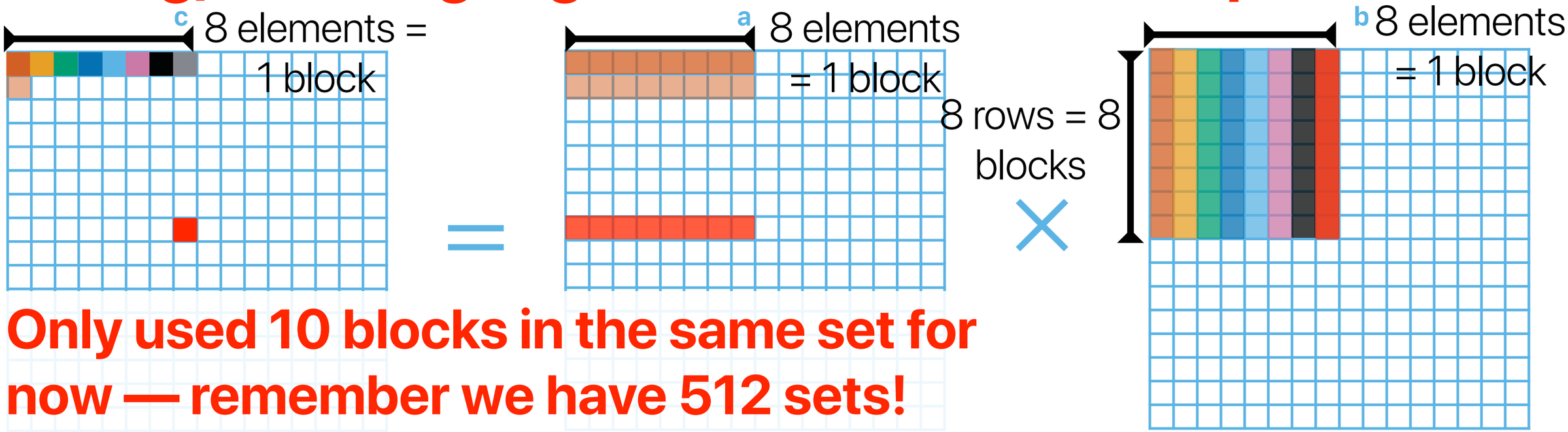
$$c_{i,j} = \sum_{k=0}^{k=N-1} a_{i,k} \times b_{k,j} = \sum_{k=0}^{k=\frac{N}{2}-1} a_{i,k} \times b_{k,j} + \sum_{k=\frac{N}{2}}^{k=N-1} a_{i,k} \times b_{k,j}$$

$$= \sum_{k=0}^{k=\frac{N}{4}-1} a_{i,k} \times b_{k,j} + \sum_{k=\frac{N}{4}}^{k=\frac{N}{2}-1} a_{i,k} \times b_{k,j} + \sum_{k=\frac{N}{2}}^{k=\frac{3N}{4}-1} a_{i,k} \times b_{k,j} + \sum_{k=3N4-1}^{k=N-1} a_{i,k} \times b_{k,j}$$

$$\vdots$$

**Let's break up the multiplications and accumulations into something fits in the cache well**

# Tiling/Blocking Algorithm for Matrix Multiplications



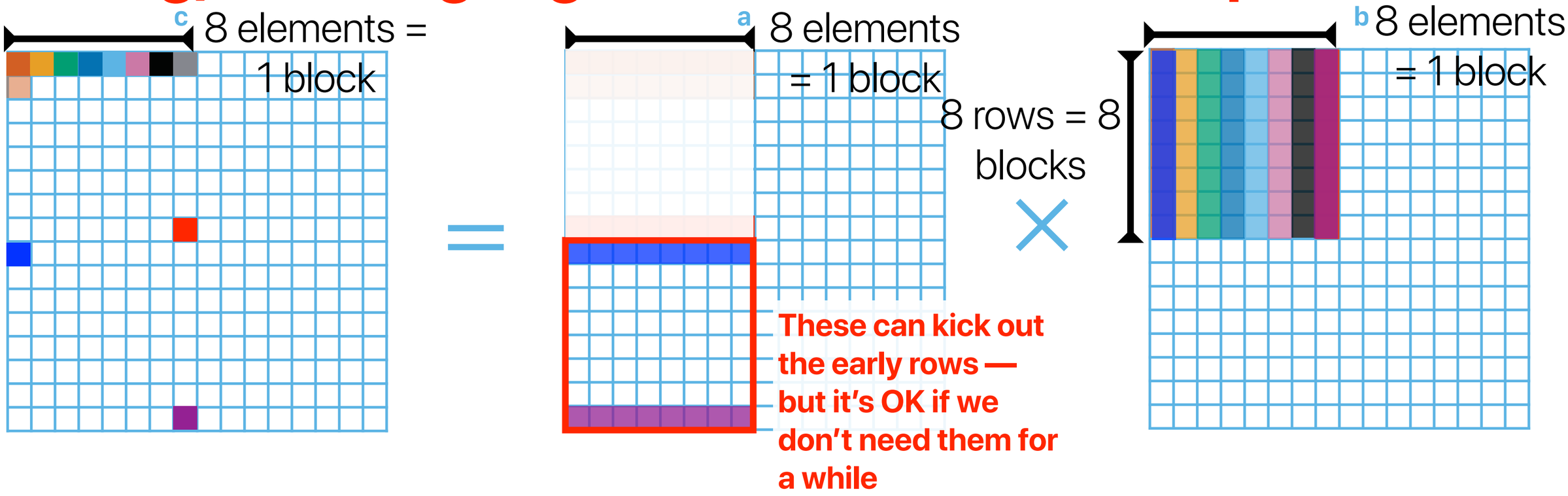

**Only used 10 blocks in the same set for now — remember we have 512 sets!**

**These are still around when we move to the next row in the "tile"**

**Only compulsory misses —**

$$miss\_rate = \frac{total\ misses}{total\ accesses} = \frac{8+8+8}{3\times 8\times 8\times 8} = 0.015625$$

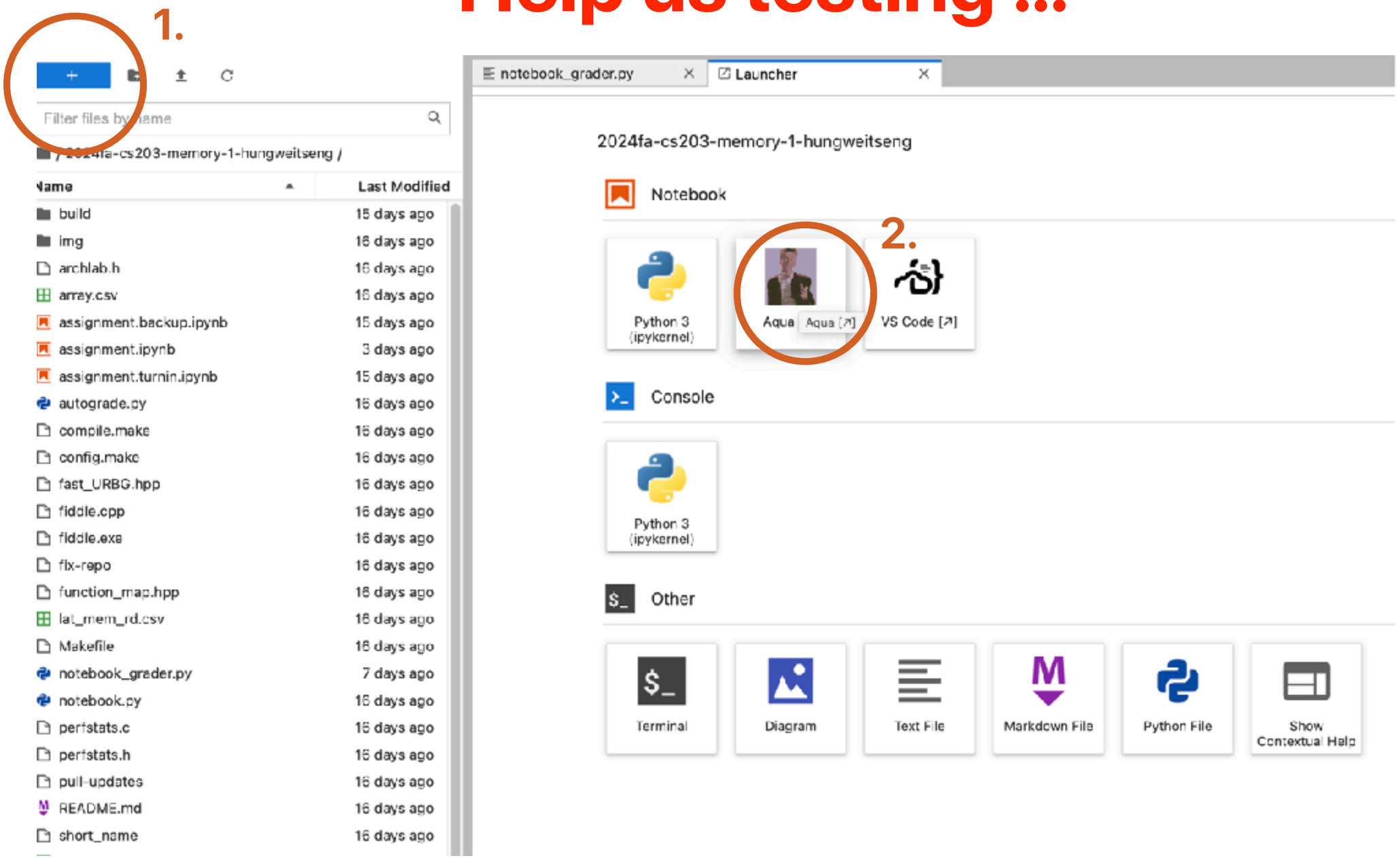
# Tiling/Blocking Algorithm for Matrix Multiplications



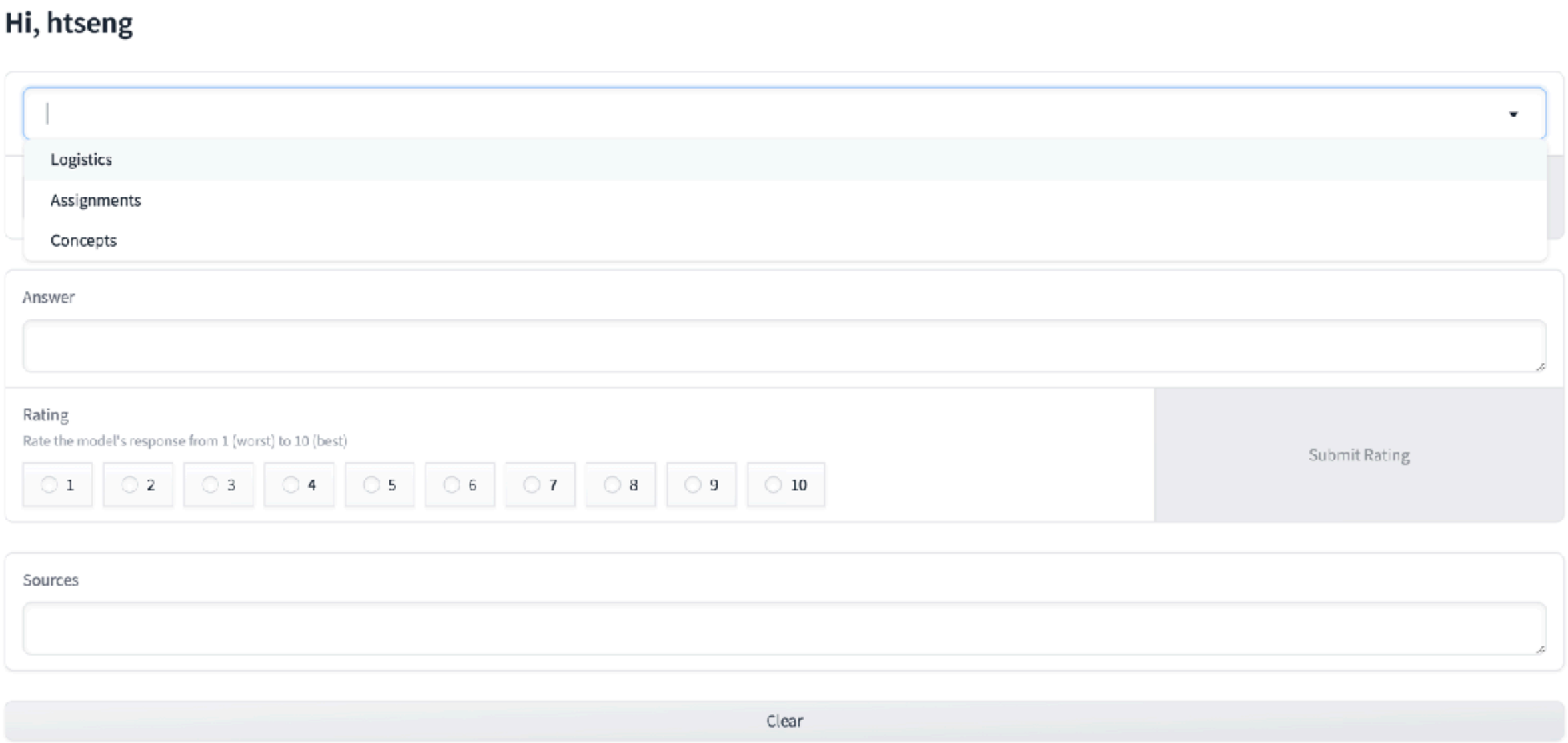

## Bringing miss rate even further lower now —

$$miss\_rate = \frac{total\ misses}{total\ accesses} = \frac{8 + 2 \times 8 + 8}{2 \times 3 \times 8 \times 8 \times 8} = 0.0104$$

# Help us testing ...



# Try to ask “real” questions you have in mind!



# Announcement

- Reading quiz #5 due **next Tuesday** before the lecture
- Assignment #2 due **this Thursday**
- Assignment #3 due **next Thursday**
- Programming Assignment #2 **due 11/7**
- Ask at least 10 questions to Aqua before the end of the quarter to receive a full credit reading quiz

Computer
Science &
Engineering

203

つづく