The Memory Hierarchy

Introduction to
Systems Software

wilkie

Spring 2019/2020

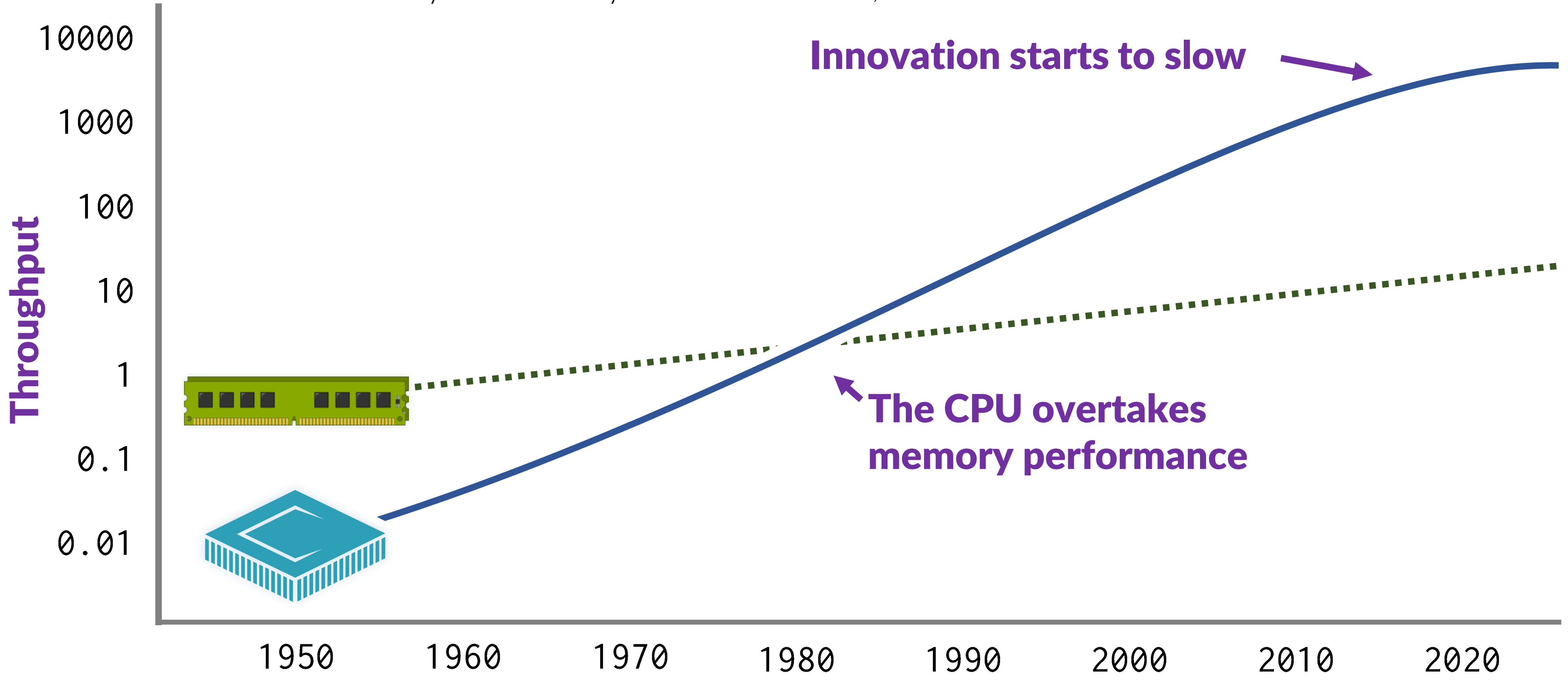
This is a Pyramid Scheme

But this knowledge is a safe investment.

Wanting Moore and Moore

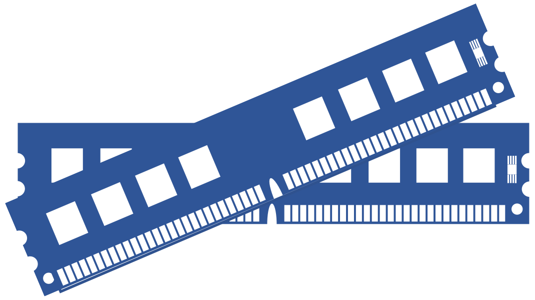
Processors and memory work together but improve at different rates.

Memory was initially faster than CPU, but its innovation was slower.

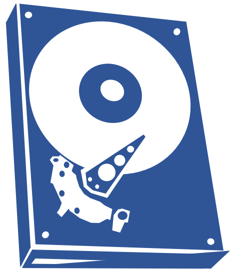


Cost of DRAM/Disk in 2020

• 8GiB	$35 - 70	• 1TiB	$50 – 80
• 16GiB	$70 - 100	• 8TiB	$200 - $300
• 32GiB	$140 - 300	• 16TiB	$400 - 500

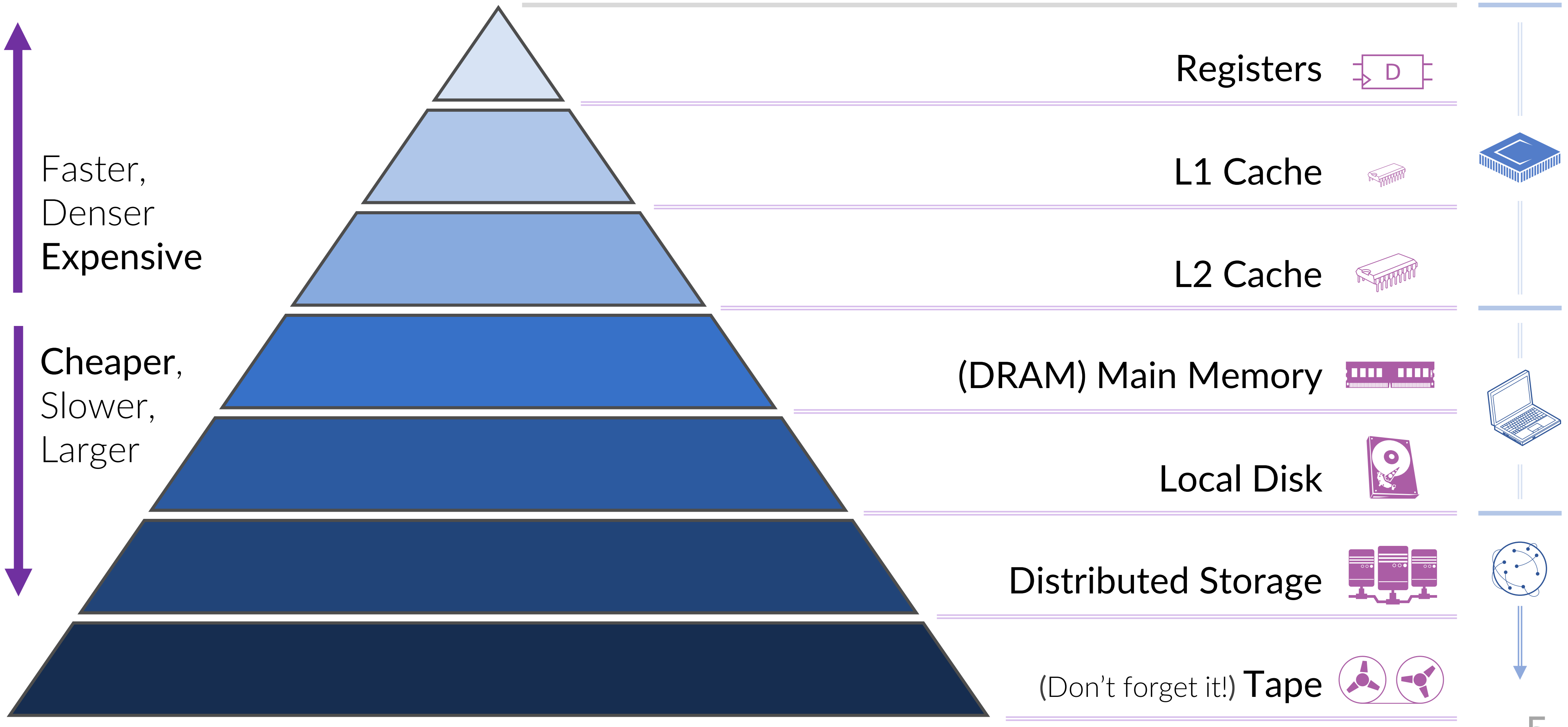


nanoseconds

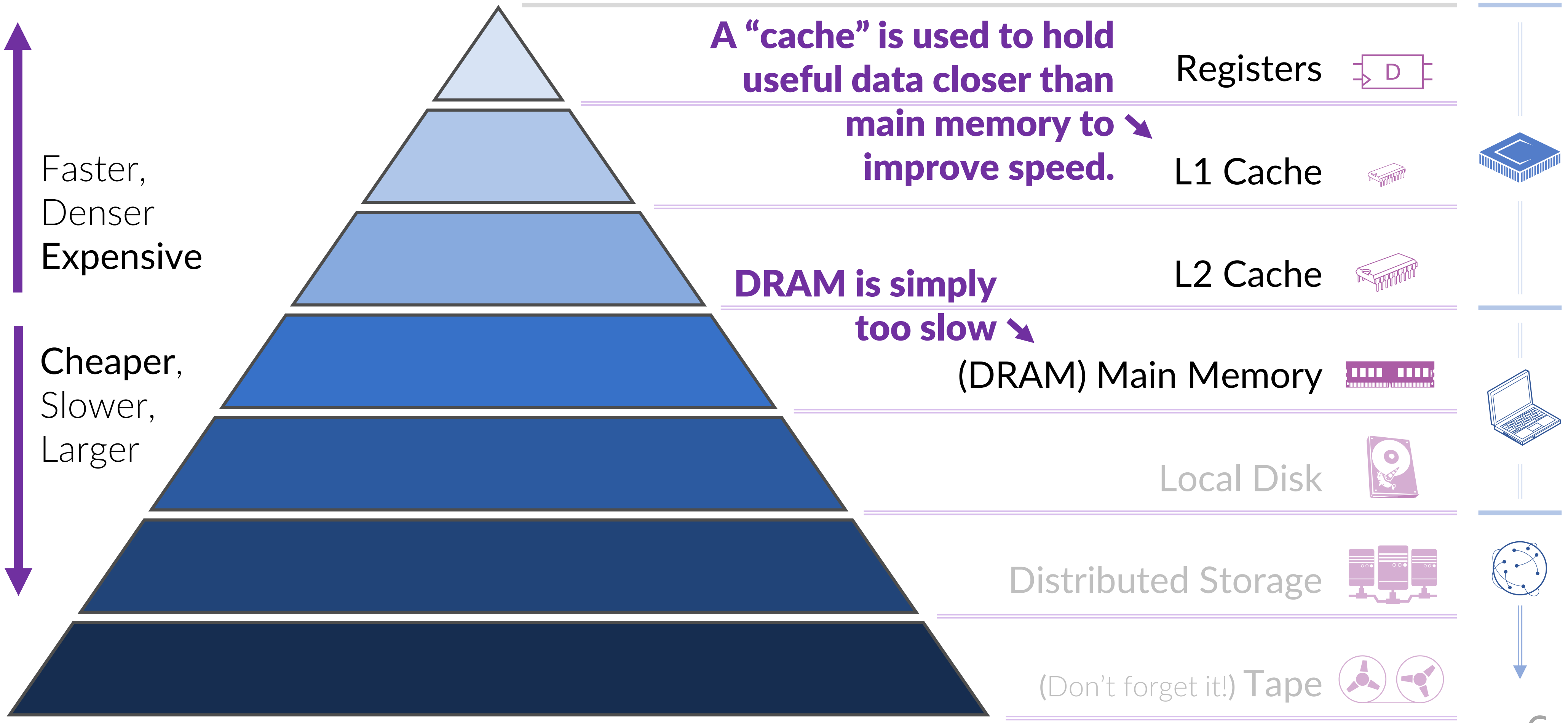


milliseconds

The memory hierarchy



The hierarchy of speed



Memory Caching

Cache: Another thing us teachers could really use more of.

Experiment: Scientific Maths

```c
#include <stdlib.h>  // for malloc

#define LIMIT (2048 * 10)

int main(void) {
  int i, j;
  int* src[LIMIT];
  int* dst[LIMIT];

  for (i = 0; i < LIMIT; i++) {
    src[i] = malloc(sizeof(int) * LIMIT);
    dst[i] = malloc(sizeof(int) * LIMIT);
  }

  for (j = 0; j < LIMIT; j++) {
    for (i = 0; i < LIMIT; i++) {
      dst[j][i] = src[j][i];
    }
  }

  return 0;
}
```

```
> time ./locality-test-A

real    0m1.803s
user    0m1.123s
sys     0m0.677s
```

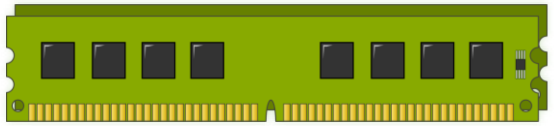
```c
#include <stdlib.h>  // for malloc

#define LIMIT (2048 * 10)

int main(void) {
  int i, j;
  int* src[LIMIT];
  int* dst[LIMIT];

  for (i = 0; i < LIMIT; i++) {
    src[i] = malloc(sizeof(int) * LIMIT);
    dst[i] = malloc(sizeof(int) * LIMIT);
  }

  for (j = 0; j < LIMIT; j++) {
    for (i = 0; i < LIMIT; i++) {
      dst[i][j] = src[i][j];
    }
  }

  return 0;
}
```

```
> time ./locality-test-B

real    0m20.374s
user    0m19.429s
sys     0m0.898s
```

Practical Performance

- **Caching is necessary for the utility of computers.**
  - The CPU/Memory gap increases (The Memory Wall)
- **In order to actually <u>use</u> these fast CPUs, we need to improve the apparent speed of RAM.**
  - Programs use memory a whole lot.
  - The bottleneck would grind performance to the point where CPUs cannot improve.

"That's a nice CPU you have there… it'd be terrible if something were to happen to it."

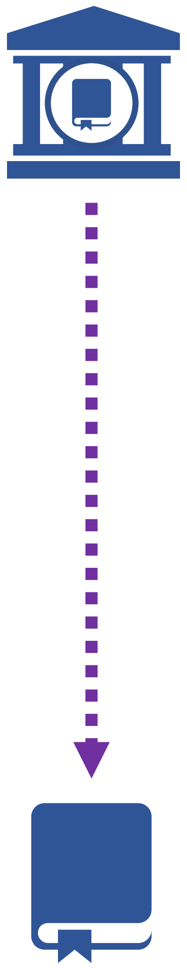


The problem: data is faaaaar away

- Let's say you want to read a book.

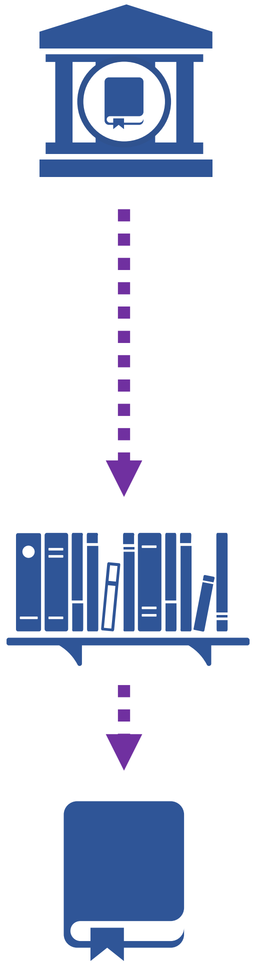
- You check it out of the library.
  - You have to go there.
  - Find the book.
  - Maybe take the bus back.
    - Wait in traffic.

- Now it sits on your desk.
  - As long as it is near you, it's easy to access the information.
  - Yet, if you need another book…
    - You would take the book ALL THE WAY back! (bare with me)



Caching: Keeping things close

- **Let's say you want to read a book.**
  - It's not on your bookshelf.
- **So, still have to check it out of the library.**
  - You gotta go there. Find the book. Etc.
  - Take the bus back.
- **Now it sits on your desk.**
  - As long as it is near you, it is easy to access the information.
  - When we need another book… we put it aside.
    - Maybe a bookshelf.
  - The next time we need it, it will be nearby.



The metaphorical cache

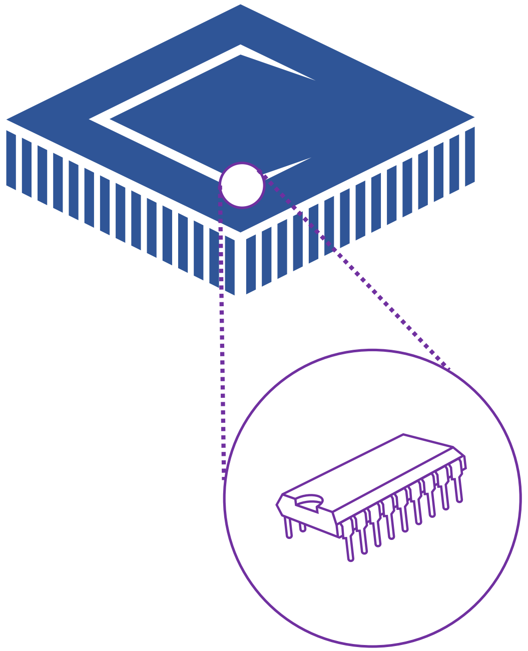
- The bookshelf is a cache.
  - It holds information that you might want later.

- It is [much] smaller than a library, but faster to retrieve things.

- However, it is small. Placing a new book on the shelf may require taking an old book off.

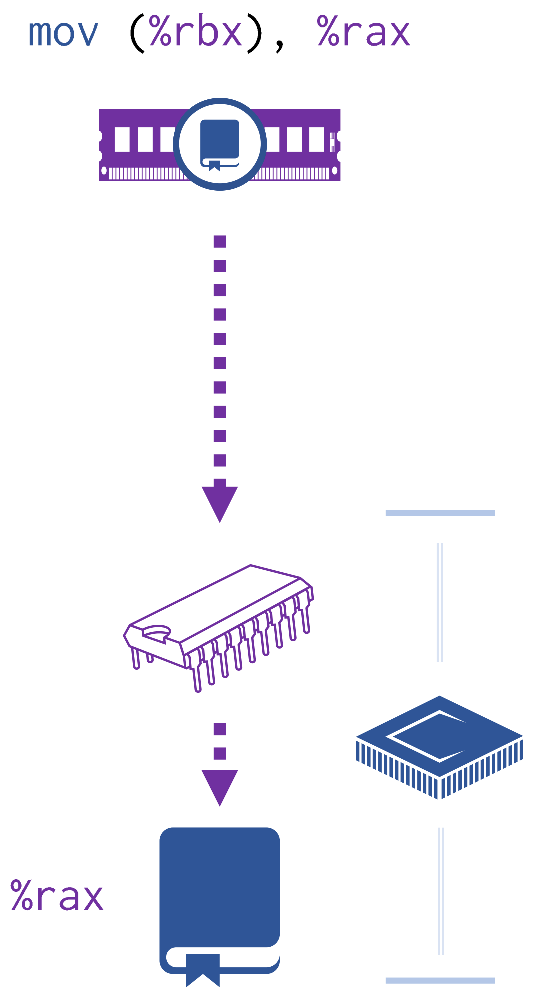
Memory cache (CPU)

- RAM is the library. It is far away and getting stuff from there is slow.
- To better handle the performance gap between the CPU and memory we add a smaller, fast memory near the CPU.
- This is the CPU cache.



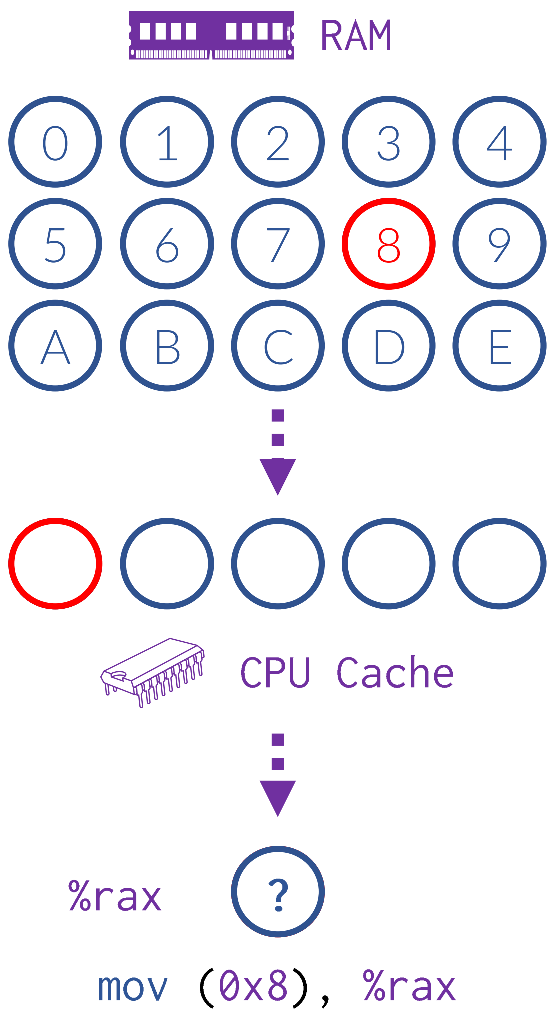
Data, the journey

- When data is requested, the goal is to read a word into a CPU register.
- The CPU first contacts the cache and asks if it has a copy.
  - If it does… that is a cache hit, and, well, that was easy. Just copy that value into the register.
- If it does not, this is a cache miss.
  - It will then contact the next component in the memory hierarchy. (RAM)
- Ram *copies* the value to cache, and the cache *copies* the value to the register.



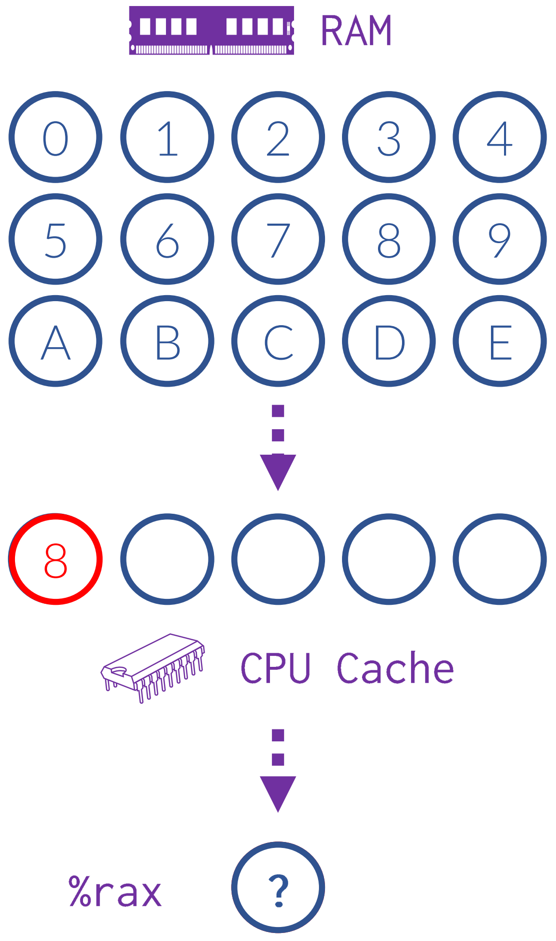
Missing the mark

- When the CPU requests memory in an empty cache, the data obviously won't be available locally.
- This is a **compulsory miss**, a "miss" due to the first access of a block of data.
  - Also known as a "cold miss."
- These are, as they suggest, completely unavoidable.
  - They always incur the high penalty associated with a memory read.



Hitting the target

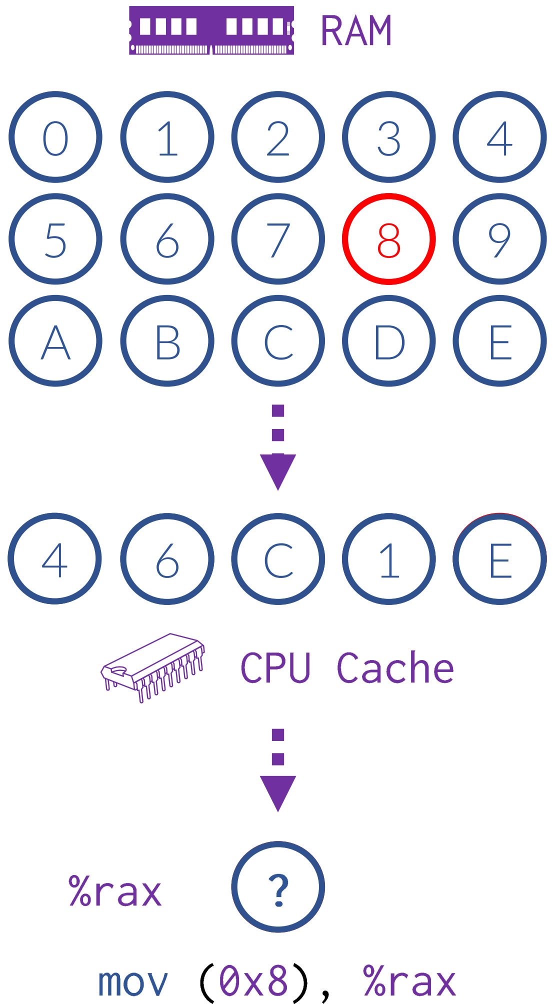
- When the CPU requests memory that happens to already be in the cache, the data is read locally (quickly).
- This is a **cache hit**.
  - Your best-case scenario.
- These avoid having to communicate at all with memory.
  - No penalty taken for reading/writing to memory.
  - Very cheap in terms of time.



```
mov (0x8), %rax
```

A cache half full...

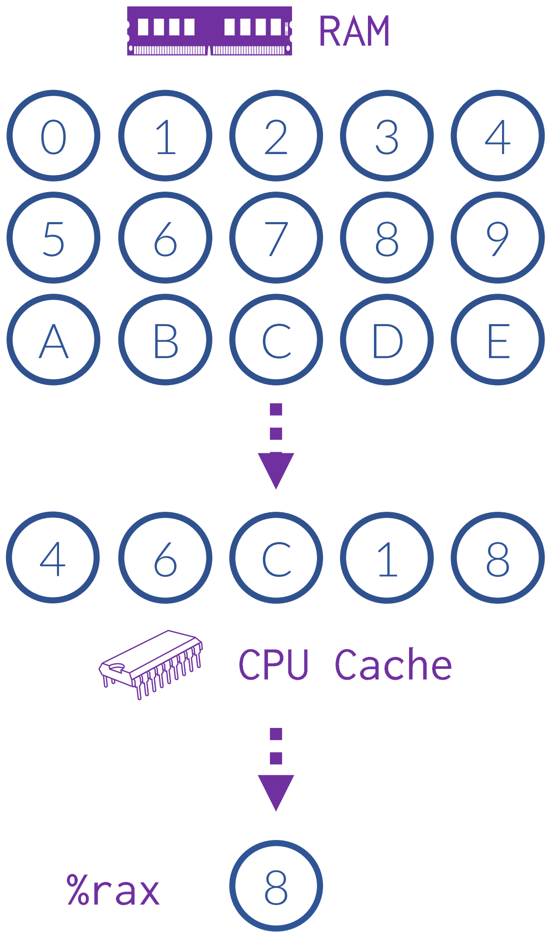
- As the CPU requests memory, the cache will fill to satisfy each compulsory miss.
- When it fills up completely, it will have no further room for the next miss.
- On a miss, it requests the data from memory.
  - Yet, where does it go?? We must remove one.
- This is a **capacity miss**. The memory requirements of the program are larger than the cache.



```
mov (0x8), %rax
```

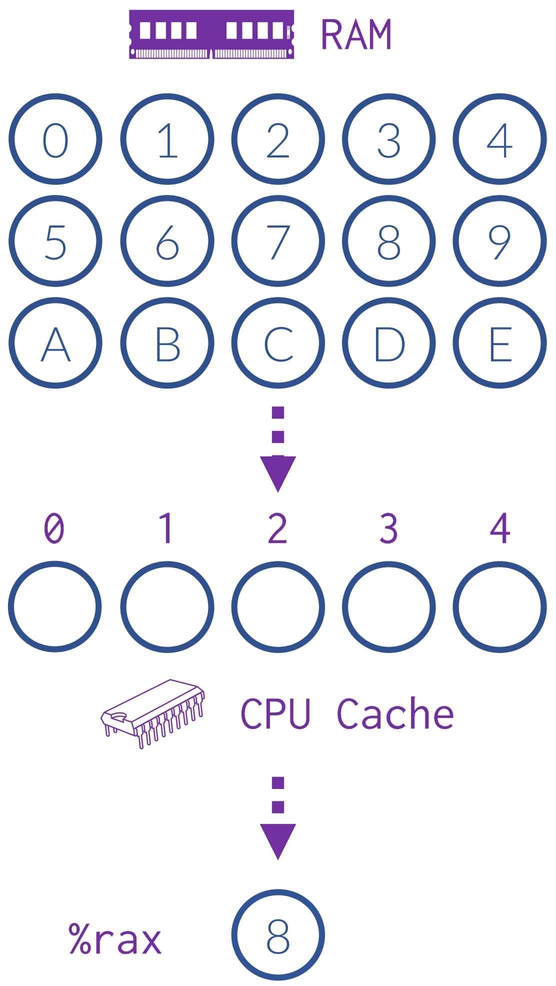
Looking closer…

- It is difficult to know what block of data to omit from this cache on such a miss.
  - However we can exploit the common locality patterns of programs to improve our cache.
- There is **temporal locality**: accessed data is likely to be used again in near future.
  - This is what caches generally capture.
- However, **spatial locality** is also likely: data is often grouped together.
  - When we access a struct field, we will often access another which is nearby in memory.



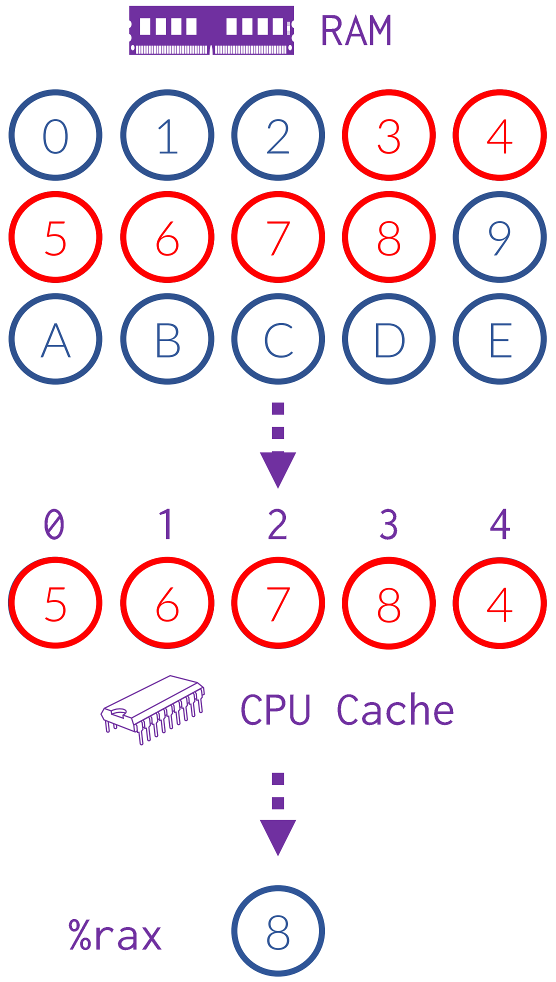
Exploring space...

- We would like to keep data that is adjacent in memory in the cache, together, at the same time.
- To do this, we “hash” the address. This is used to determine the cache slot.
  - Just a fancy way to say: we divide the address by the cache size and use the remainder.
- Every 0th block, 1st block, 2nd block, etc.
- The 5th block (in this example) goes to the `0` slot, the 6th goes to the `1` slot, and so on.



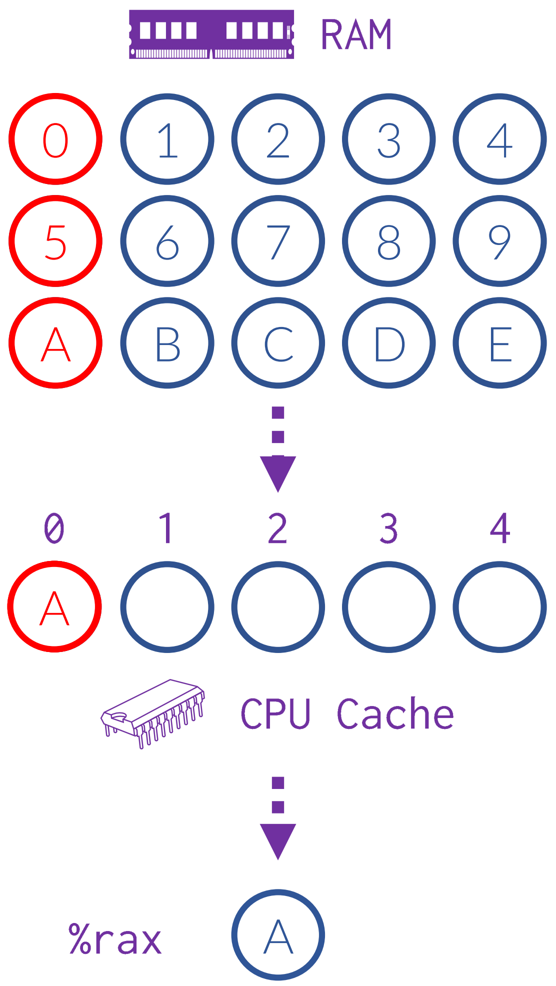
Direct and to the point...

- Let's read addresses 3, 4, 5, 6, and 7 (in that order) from memory.
- Reading address 8 next incurs a capacity miss, but it evicts the address that is furthest away from the others.
  - This type of cache is good for programs that read through data sequentially.
  - That is because such programs will always remove the least recently used block on a miss, as shown here.
- Because every address has a specific cache slot, this is called a **direct-mapped cache**.



Missing your connection...

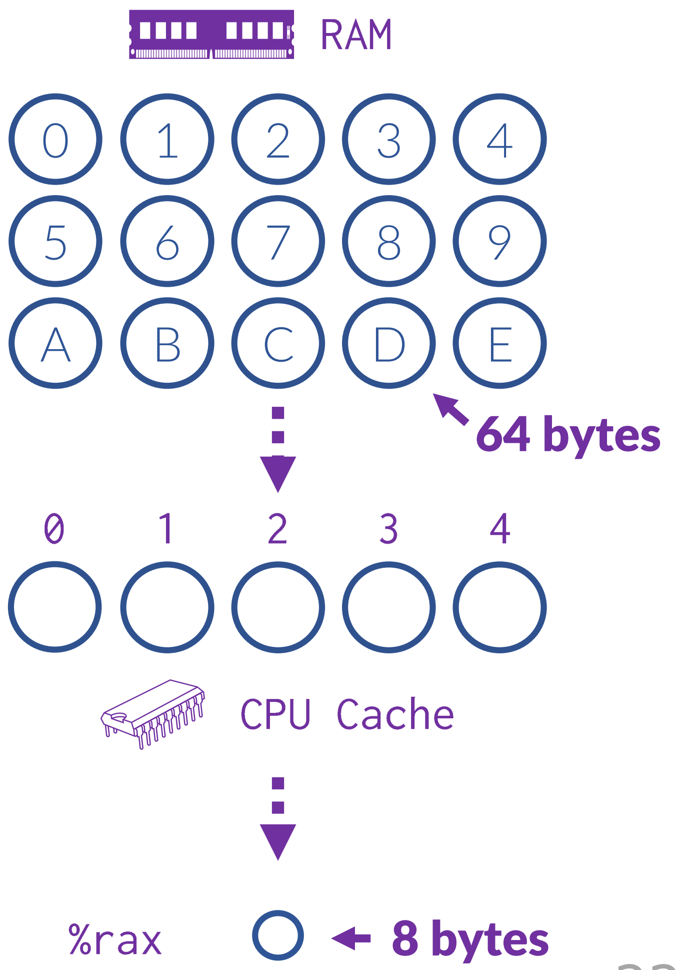
- Let’s consider an antagonist pattern.
  - What is the worst case for this cache?
- If we read every 5th address in our memory in order, we would overwhelm our direct-mapped cache.
  - Let’s access 0, 5, A in that order.
- Accessing address 0 is a <u>compulsory miss</u>.
  - Address 5, however, is a miss.
  - But our cache isn’t full!!
- A miss that occurs even though your cache *could* fit the block is called a **conflict miss**.



How big is that block?

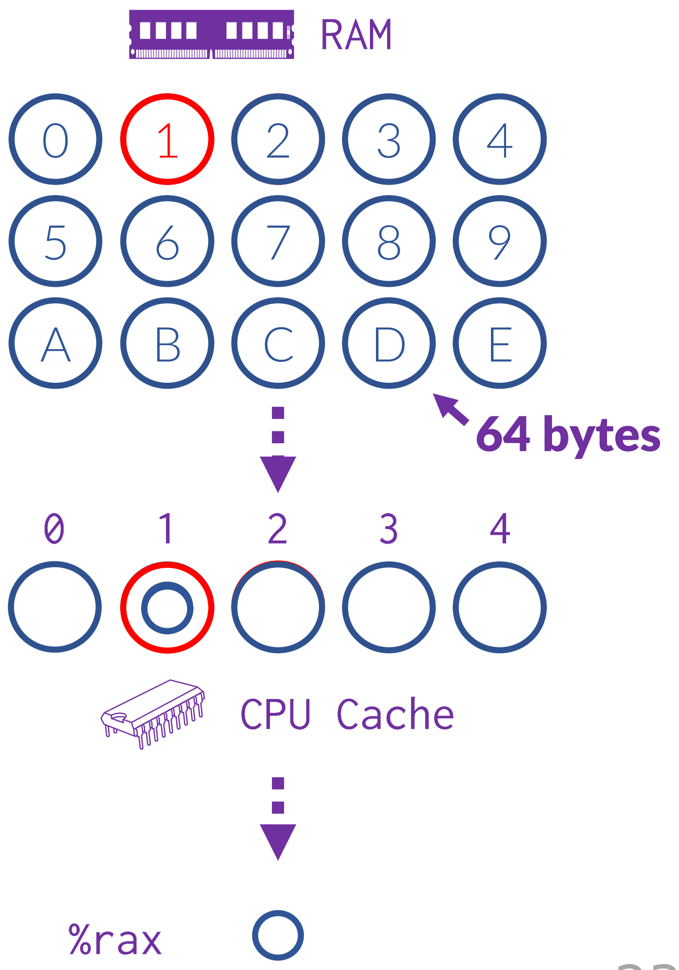
- **Spatial locality is SO prevalent that it makes a whole lot of sense to pull more data than is requested.**
  - If we request a word (8 bytes) from memory, and we have a cache miss, let's pull 8 words at a time (64 bytes).

- **Therefore, the blocks visualized to the right can have a size, called the block size.**
  - The bigger the block, the better spatial locality will become.
  - However, the more time it takes to copy from memory and the higher penalty if you throw it away on a miss!



Block size helps locality

- When we request an address from our cache, we are requesting the block that contains that address.
  - Here, Block 0 contains byte addresses 0x00 through 0x39. Block 1 is 0x40 to 0x79, etc.
- Let's request 64-bit words in order starting at address 0x40 (Block 1)
  - There are 8 words in each cache block.
  - Therefore, we have only one compulsory miss.
  - And then we have 7 cache hits!!
- If we request the ninth word, we will be at address 0x80 (and a compulsory miss.)



Once again... A Tale of Two C... um... programs

```c
#include <stdlib.h>  // for malloc

#define LIMIT (2048 * 10)

int main(void) {
  int i, j;
  int* src[LIMIT];
  int* dst[LIMIT];

  for (i = 0; i < LIMIT; i++) {
    src[i] = malloc(sizeof(int) * LIMIT);
    dst[i] = malloc(sizeof(int) * LIMIT);
  }

  for (j = 0; j < LIMIT; j++) {
    for (i = 0; i < LIMIT; i++) {
      dst[j][i] = src[j][i];
    }
  }

  return 0;
}
```

Allocates matrices. (Array of arrays)

Copies one matrix to another. (data itself is uninitialized.)

```
> time ./locality-test-A

real    0m1.803s
user    0m1.123s
sys     0m0.677s
```

```c
#include <stdlib.h>  // for malloc

#define LIMIT (2048 * 10)

int main(void) {
  int i, j;
  int* src[LIMIT];
  int* dst[LIMIT];

  for (i = 0; i < LIMIT; i++) {
    src[i] = malloc(sizeof(int) * LIMIT);
    dst[i] = malloc(sizeof(int) * LIMIT);
  }

  for (j = 0; j < LIMIT; j++) {
    for (i = 0; i < LIMIT; i++) {
      dst[i][j] = src[i][j];
    }
  }

  return 0;
}
```

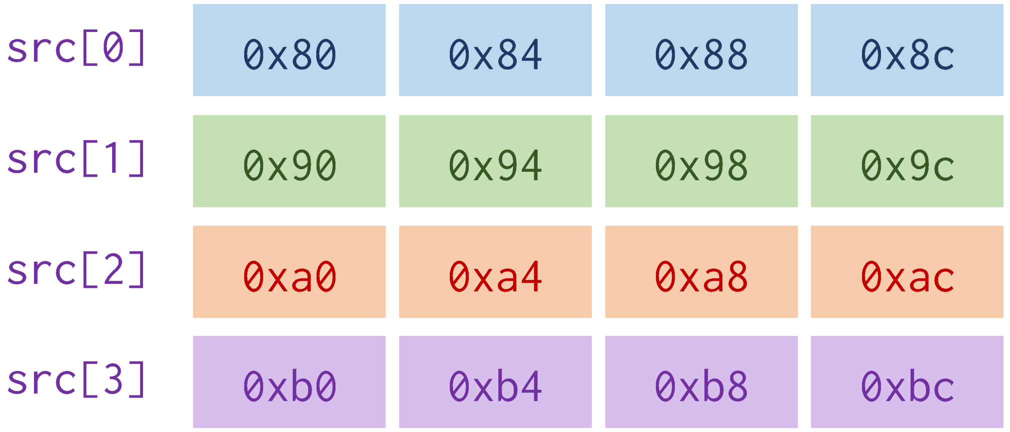
Iterates through column. (Other code goes through row)

```
> time ./locality-test-B

real    0m20.374s
user    0m19.429s
sys     0m0.898s
```

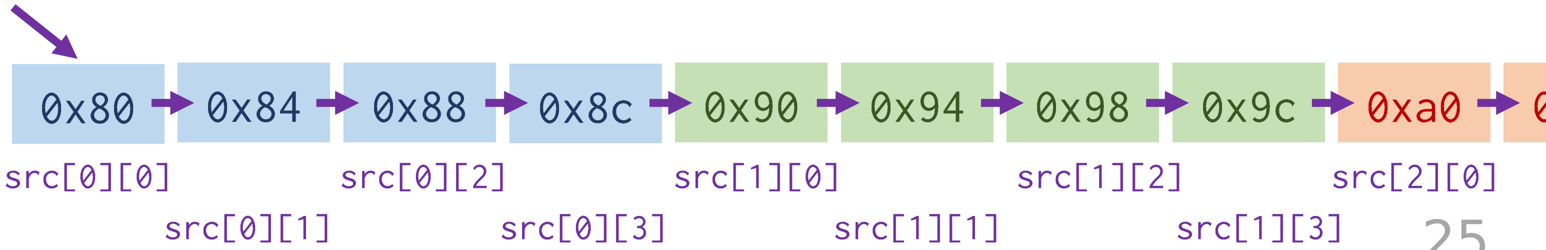
Once again... A Tale of Two C... um... programs

```c
#include <stdlib.h>  // for malloc

// We will look at a 4 by 4 matrix

int main(void) {
  int i, j;
  int* src[4];
  int* dst[4];

  for (i = 0; i < 4; i++) {
    src[i] = malloc(sizeof(int) * 4);
    dst[i] = malloc(sizeof(int) * 4);
  }

  for (j = 0; j < 4; j++) {
    for (i = 0; i < 4; i++) {
      dst[j][i] = src[j][i];
    }
  }

  return 0;
}
```

- We will simplify by looking at a 4x4 matrix.
  - We want to get the addresses being used to see the access pattern. (Goes across row)



src[0]	0x80	0x84	0x88	0x8c
src[1]	0x90	0x94	0x98	0x9c
src[2]	0xa0	0xa4	0xa8	0xac
src[3]	0xb0	0xb4	0xb8	0xbc



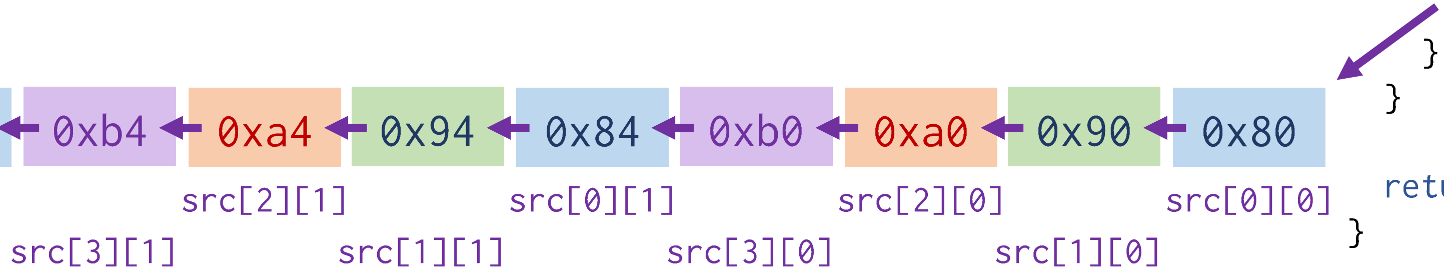
$$\begin{tikzcd}
\text{0x80} \arrow[r] & \text{0x84} \arrow[r] & \text{0x88} \arrow[r] & \text{0x8c} \arrow[r] & \text{0x90} \arrow[r] & \text{0x94} \arrow[r] & \text{0x98} \arrow[r] & \text{0x9c} \arrow[r] & \text{0xa0} \arrow[r] & \text{0}
\end{tikzcd}$$

Once again... A Tale of Two C... um... programs

- We will simplify by looking at a 4x4 matrix.
  - Notice the different type of access pattern.
  - (Goes down the column)

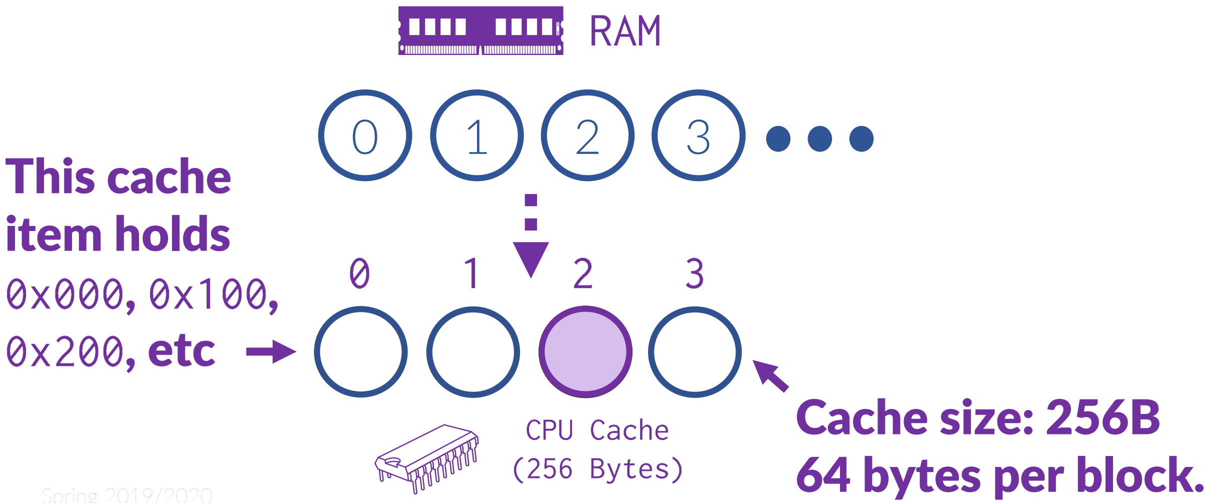
src[0]	0x80	0x84	0x88	0x8c
src[1]	0x90	0x94	0x98	0x9c
src[2]	0xa0	0xa4	0xa8	0xac
src[3]	0xb0	0xb4	0xb8	0xbc

```c
#include <stdlib.h>  // for malloc

// We will look at a 4 by 4 matrix

int main(void) {
  int i, j;
  int* src[4];
  int* dst[4];

  for (i = 0; i < 4; i++) {
    src[i] = malloc(sizeof(int) * 4);
    dst[i] = malloc(sizeof(int) * 4);
  }

  for (j = 0; j < 4; j++) {
    for (i = 0; i < 4; i++) {
      dst[i][j] = src[i][j];
    }
  }

  return 0;
}
```



The Antagonist

- One program reads words sequentially in memory (good spatial locality)
  - The other reads each word as far apart as possible! (worst spatial locality)
- Let's look at making the matrices much larger! Let's make each row span 256 Bytes. (4 blocks, which is the size of our cache!)



0x80	Miss	0x80	Miss
0x84	Hit	0x180	Miss
0x88	Hit	0x280	Miss
0x8c	Hit	0x380	Miss
0x90	Hit	0x84	Miss
0x94	Hit	0x184	Miss
0x98	Hit	0x284	Miss
0x9c	Hit	0x384	Miss
0xa0	Hit	0x88	Miss
0xa4	Hit	0x188	Miss
0xa8	Hit	0x288	Miss

Cache Performance

- Recall that caches make computers practical.
  - Why? Well…
  - Our “slow” program effectively did not use cache, and it was 10 times slower.

- Simply: Caches offer much faster accesses than DRAM.
  - Perhaps 100s of times faster.

- Consider the math:
  - **miss rate** (MR): Fraction of memory accesses not in cache.
  - **hit rate** (HR): Fraction of memory accesses found in cache. ($HR = 1 - MR$)
  - **hit time** (HT): Time it takes to read a block from cache to CPU. (Best case)
  - **miss penalty** (MP): Time it takes to read from main memory to cache.

Cache Performance

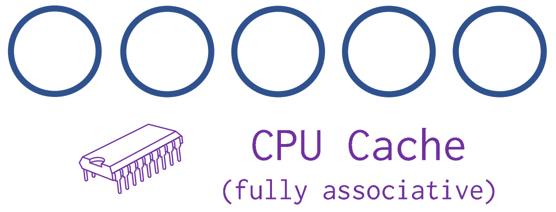
- Recall that caches make computers practical.
- Consider the math:
  - **miss rate** (MR): Fraction of memory accesses not in cache.
  - **hit rate** (HR): Fraction of memory accesses found in cache. ($\mathrm{H}R = 1 - MR$)
  - **hit time** (HT): Time it takes to read a block from cache to CPU. (Best case)
  - **miss penalty** (MP): Time it takes to read from main memory to cache.
- **Average Memory Access Time** (AMAT): The time it takes, on average, to perform a memory request, considering the cache performance.
  - $AMAT = HT + MR \times MP$
- Assuming a HT of 1 clock cycle and a MP of 100 clock cycles…
  - A HR of 97%: $AMAT = 1 + 0.03 \times 100 = 4\ cycles$
  - A HR of 99%: $AMAT = 1 + 0.01 \times 100 = 2\ cycles$
  - A hit-rate jump from just 97% to 99% *doubles memory performance*. Wow.

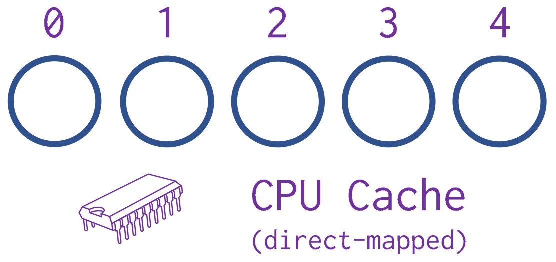
Cache Layout Summary

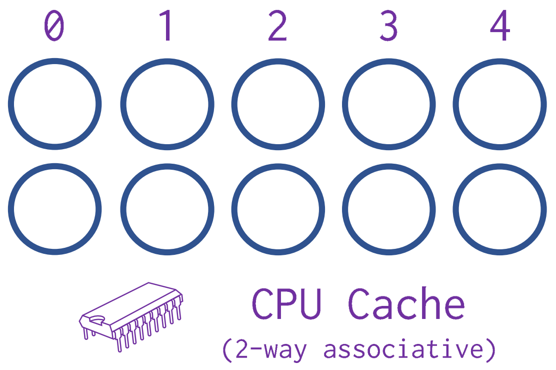
- We have seen two types of cache layouts.
  - A freeform cache: blocks go wherever. ¯_(ツ)_/¯
    - Also called a **fully associative cache.**
  - A direct-mapped cache: blocks go into slots.

- They have their own trade-offs, and as usual…
  - We can have a hybrid approach!

- Here, each cache slot has multiple bins.
  - You only need to evict when you fill up the bins. Best of both worlds!
  - Which do you evict? (Hmm… difficult choice.)



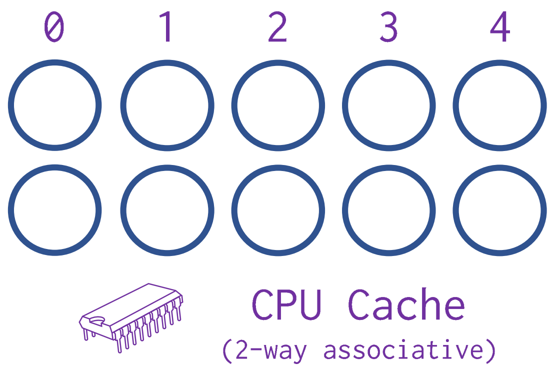


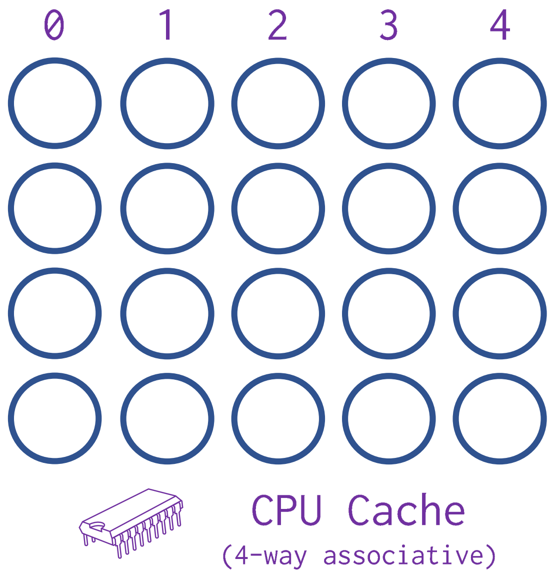


Associativity

- **With an associative cache, the address determines the slot.**
  - Much like a direct-mapped cache.
  - However, the slot has a number of bins.
  - Any bin in the slot is viable for a block.
  - The number of bins is the number of "ways"
    - A direct-mapped cache is a 1-way cache.

- **When the cache determines if the block is in the cache already…**
  - It determines the slot.
  - It scans every bin for a block tagged with that exact address.
  - Therefore, the cache performance degrades as you increase the number of ways.





Summary

- The notion of storing data is a complicated one.
  - Different technologies have different strengths (and costs)
  - Often trade-off between:
    - fast / small, expensive
    - slow / big, cheap
  - Hardware designs attempt to accommodate a variety of technologies.
    - Often using fast/small memories to act as a "cache" for slower ones.
- Caches can be arranged in several ways:
  - Blocks go anywhere (fully-associative)
  - Blocks go in particular slots (direct-mapped / 1-way associative)
  - Hybrid: Blocks go to particular slots… but then can go in any bin in that slot.
- Caches attempt to exploit temporal and spatial locality of programs.
  - And even a slight improvement to hit rate can dramatically improve overall performance of a program!