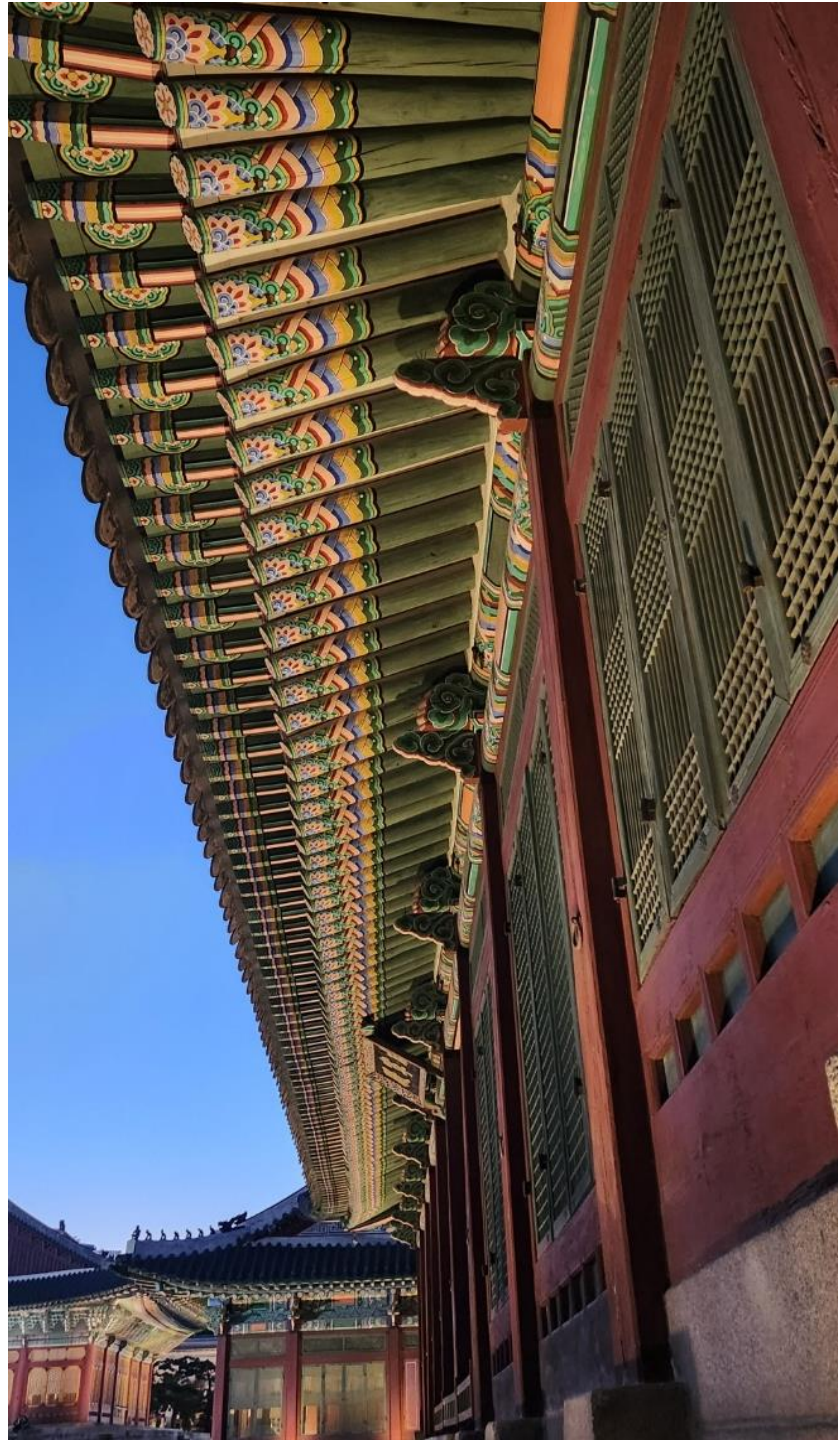
Memory Hierarchy

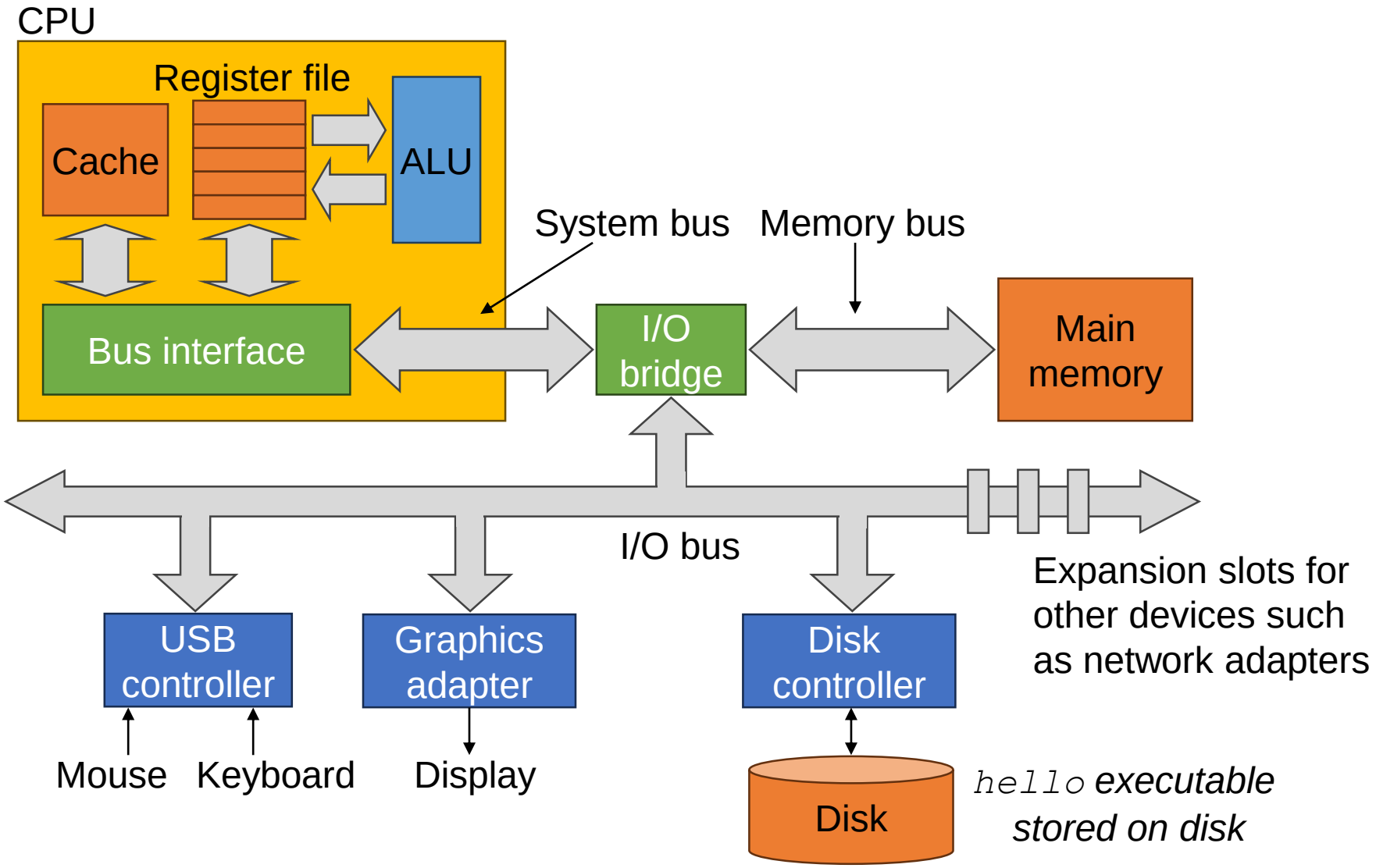
HGU



Contents

- The Memory Hierarchy
- Locality
- Caching
- Writing Cache-Friendly Code
- Cache Analysis and Valgrind

Hardware Organization of a Typical System

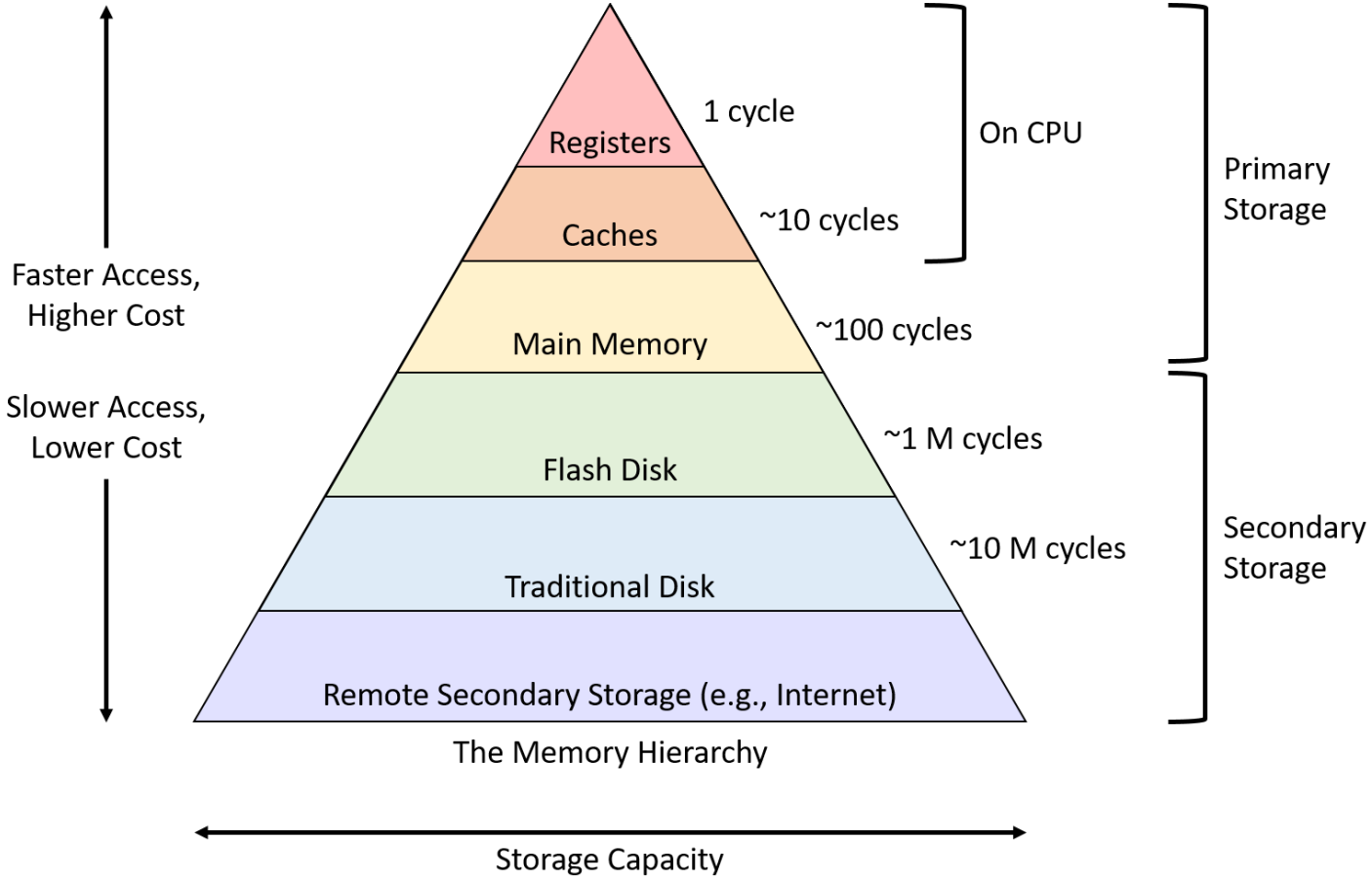


Memory Technology

- Large memory is slow and fast memory is small
- But, user wants large and fast memory
  - Static RAM (SRAM)
    - 0.5ns – 2.5ns, $2000 – $5000 per GB
  - Dynamic RAM (DRAM)
    - 50ns – 70ns, $20 – $75 per GB
  - Magnetic disk
    - 5ms – 20ms, $0.20 – $2 per GB
- Systems use devices that are fast and devices that store a large amount of data, but no single device does both

The Memory Hierarchy

- Trade-off between performance and capacity in computer memories



Performance Metrics for Memory Devices

- Capacity
  - The amount of data a device can store.
  - Measured in bytes
- Latency
  - The amount of time it takes for a device to respond with data after it has been instructed to perform a data retrieval operation.
  - Measured in second (milliseconds or nanoseconds) or CPU cycles
- Transfer rate
  - The amount of data that can be moved between the device and main memory over some interval of time.
  - Transfer rate is also known as throughput
  - Measured in bytes per second (or bits per second)

Primary Storage

- Primary storage includes registers, cache, and main memory
- Primary storage devices can be accessed directly by a program on the CPU.

Device	Capacity	Approx. latency	RAM type
Register	4 - 8 bytes	< 1 ns	SRAM (?)
CPU cache	1 - 32 megabytes	5 ns	SRAM
Main memory	4 - 64 gigabytes	100 ns	DRAM

Primary Storage: SRAM and DRAM

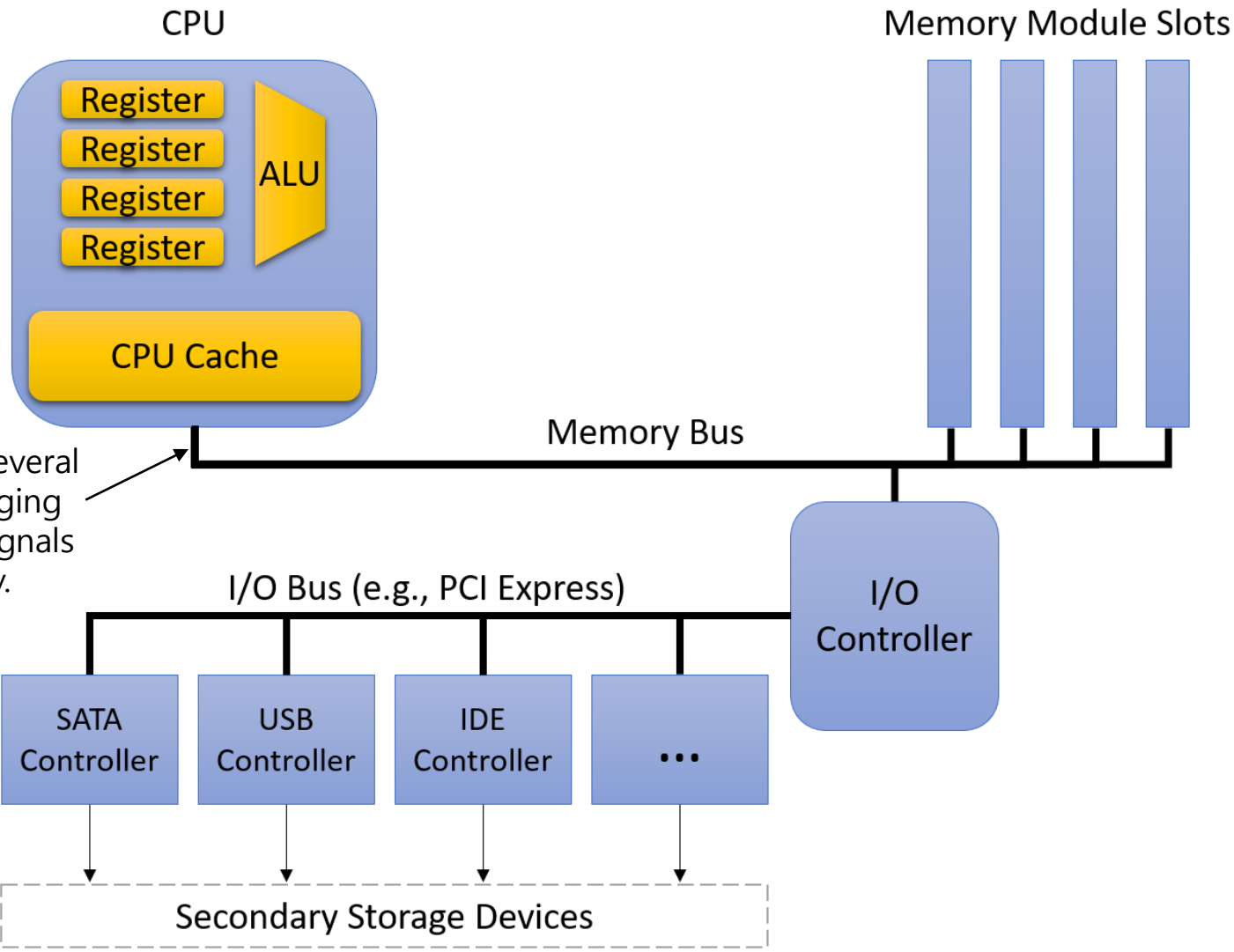
- Static RAM (SRAM) stores data in small electrical circuits.
  - It is typically the fastest type of memory and is integrated directly into a CPU to build cache.
  - It is relatively expensive in its cost to build, cost to operate (power consumption), and in the amount of space it occupies.
- Dynamic RAM (DRAM) stores data using electrical components called capacitors that hold an electrical charge.
  - It's called "dynamic" because a DRAM system must frequently refresh the charge of its capacitors to maintain a stored value.
  - Modern systems use DRAM to implement main memory on modules that connect to the CPU via a high-speed interconnect called the memory bus.

Secondary Storage

- Physically, secondary storage devices connect to a system even farther away from the CPU than main memory
- Two most common secondary storage devices today are hard disk drive (HDDs) and flash-based solid-state drives (SSDs).

Device	Capacity	Latency	Transfer rate
Flash disk	0.5 - 2 terabytes	0.1 - 1 ms	200 - 3,000 megabytes / second
Traditional hard disk	0.5 - 10 terabytes	5 - 10 ms	100 - 200 megabytes / second
Remote network server	Varies considerably	20 - 200 ms	Varies considerably

Secondary Storage and I/O Bus

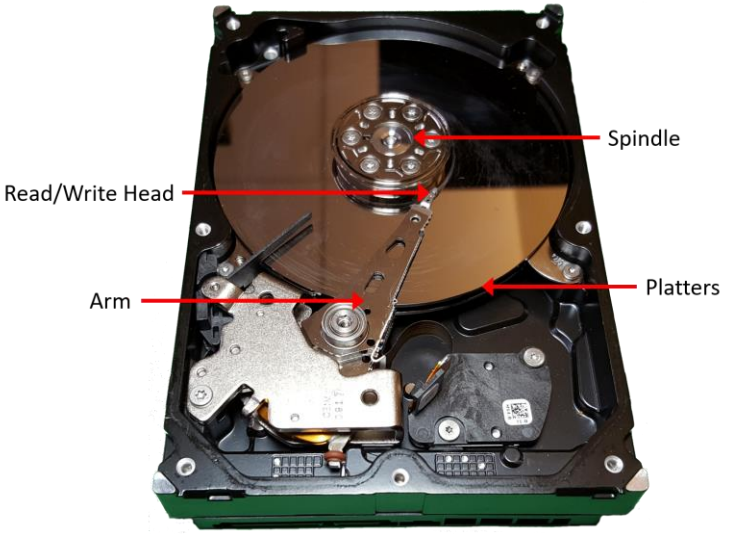


The memory bus consists of several collections of wires for exchanging addresses, data, and control signals between the CPU and memory.

Secondary Storage: HDD and SSD

- Hard Disk Drive (HDD)
  - Seek time: the arm moves the disk head to the desired track
  - Rotational latency: the disk must wait for the platter to rotate until the disk head is directly above the location that stores the desired data
- Solid-State Drive (SSD)
  - SSDs do not rely on mechanical movements, leading to lower latency (no seek time or rotational latency).
  - Flash memory: The dominant technology in SSDs, enabling faster read, write, and erase speeds compared to HDDs

<HDD>



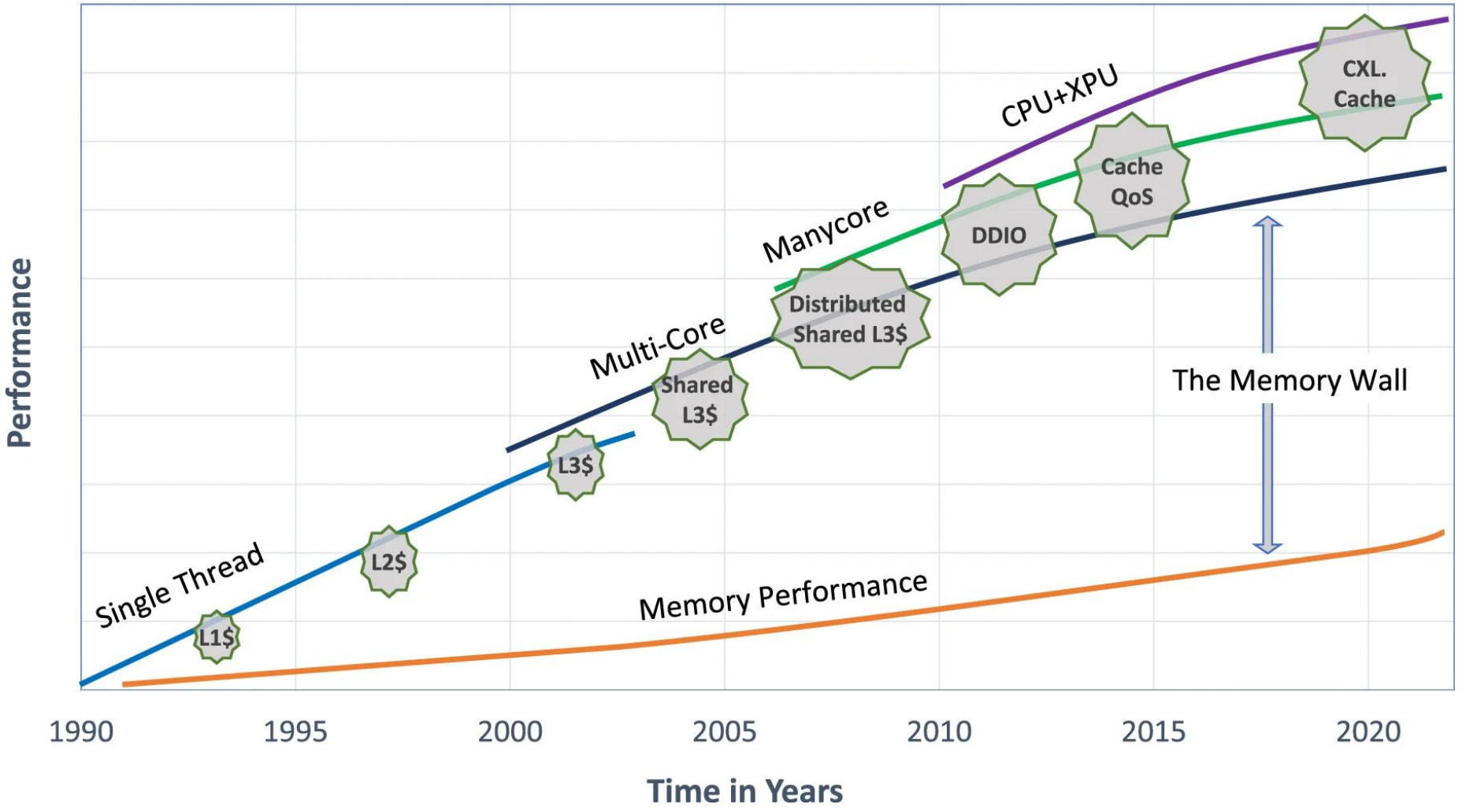
<NVMe SSD>

https://semiconductor.samsung.com/

Contents

- The Memory Hierarchy
- Locality
- Caching
- Writing Cache-Friendly Code
- Cache Analysis and Valgrind

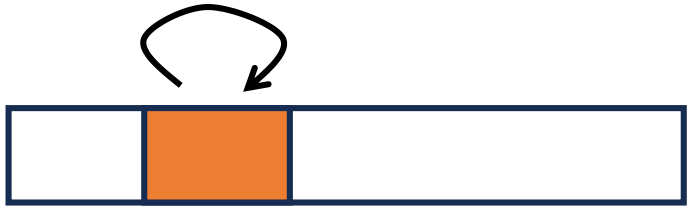
The CPU-Memory Gap



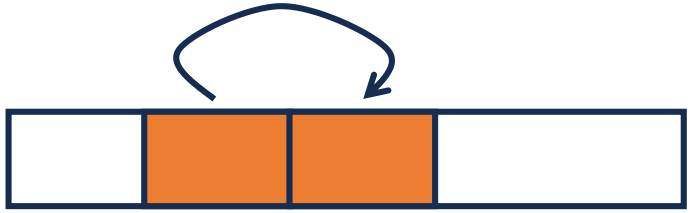
R. Iyer et al., "Advances in Microprocessor Cache Architectures Over the Last 25 Years" in IEEE Micro, vol. 41, no. 06, pp. 78-88, Nov.-Dec. 2021

Locality

- **Principle of Locality**: Programs tend to use data and instructions with addresses near or equal to those they have used recently
- Temporal locality
  - Recently referenced items are likely to be referenced again in the near future.
  - If a program has used a variable recently, it's likely to use that variable again soon.



- Spatial locality
  - Items with nearby addresses tend to be referenced close together in time.
  - If a program accesses data at addresses N and N+4, it's likely to access N+8 soon.



Locality: Example #1

```
/* Sum up the elements in an integer array of length len. */
int sum_array(int *array, int len) {
    int i;
    int sum = 0;
    for (i = 0; i < len; i++) {
        sum += array[i];
    }
    return sum;
}
```

- Temporal locality: `i`, `len`, `sum`, and `array` (the base address of the array)
- Spatial locality: `array` contents

Locality: Example #2

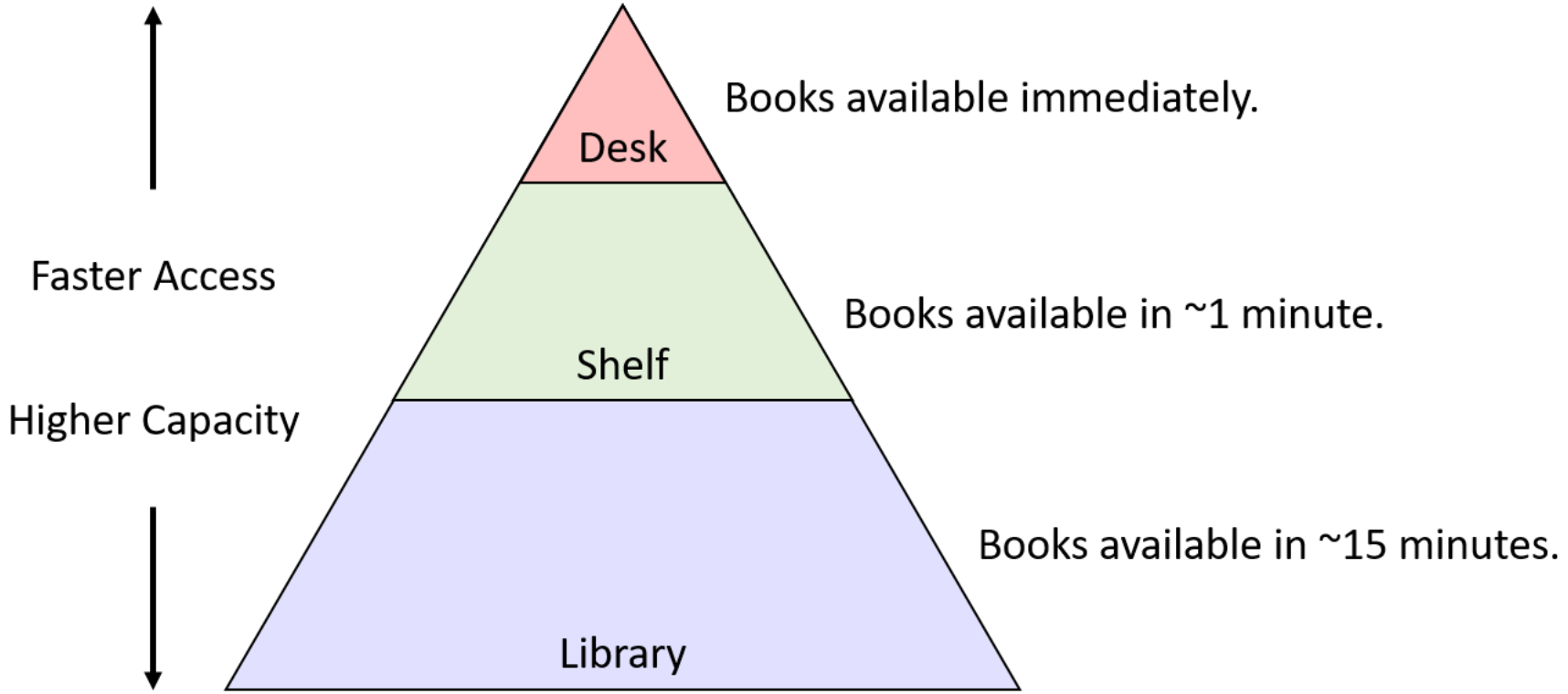
```
float averageMat_v1(int **mat, int n) {
    int i, j, total = 0;

    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            // Note indexing: [i][j]
            total += mat[i][j];
        }
    }

    return (float) total / (n * n);
}
```

```
float averageMat_v2(int **mat, int n) {
    int i, j, total = 0;

    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            // Note indexing: [j][i]
            total += mat[j][i];
        }
    }

    return (float) total / (n * n);
}
```

- Temporal locality:
- Spatial locality:

From Locality to Caches

- Example book storage hierarchy



Example Book Storage Hierarchy

From Locality to Caches: Temporal Locality

- Temporal locality suggests that frequently used books should be kept as close to the desk as possible.
  - While occasionally moving a book to the shelf is acceptable, it would be inefficient to return a book to the library if it will be needed again the next day. Conversely, a book that hasn't been used for a long time could be a good candidate for returning to the library.
- When Fiona needs a book, she moves it to the desk.
- If the desk is full, the least recently used book is moved to the shelf.
- If the shelf is full, the least recently used book on the shelf is returned to the library.

From Locality to Caches: Spatial Locality

- Spatial locality suggests that when making a trip to the library, it is beneficial to borrow multiple books at once to reduce the likelihood of future trips.
  - For example, if Fiona has a literature assignment to read Henry VI, Part I, it would be wise to also borrow Parts II and III, which are likely nearby on the shelf.
- Programs similarly store related data in memory in a contiguous manner. Constructs like arrays or structs keep related data in adjacent memory locations, creating a spatial pattern when accessing consecutive elements.
- Applying spatial locality to storage implies that when retrieving data from main memory, the system should also fetch the surrounding data.

Locality: Example #3

- Permute the loops in the following function so that it scans the three-dimensional array a with a stride-1 reference pattern.

```
int productarray3d (int a[N][N][N]) {
    int i, j, k, product = 1;

    for (i = N-1; i >= 0; i--) {
        for (j = N-1; j >= 0; j--) {
            for (k = N-1; k >= 0; k--) {
                product *= a[j][k][i];
        }
    }

    return product;
}
```

Locality: Example #4

- The following three functions perform the same operation with varying degrees of spatial locality. Rank-order the functions with respect to the spatial locality enjoyed by each. Explain how you arrived at your ranking.

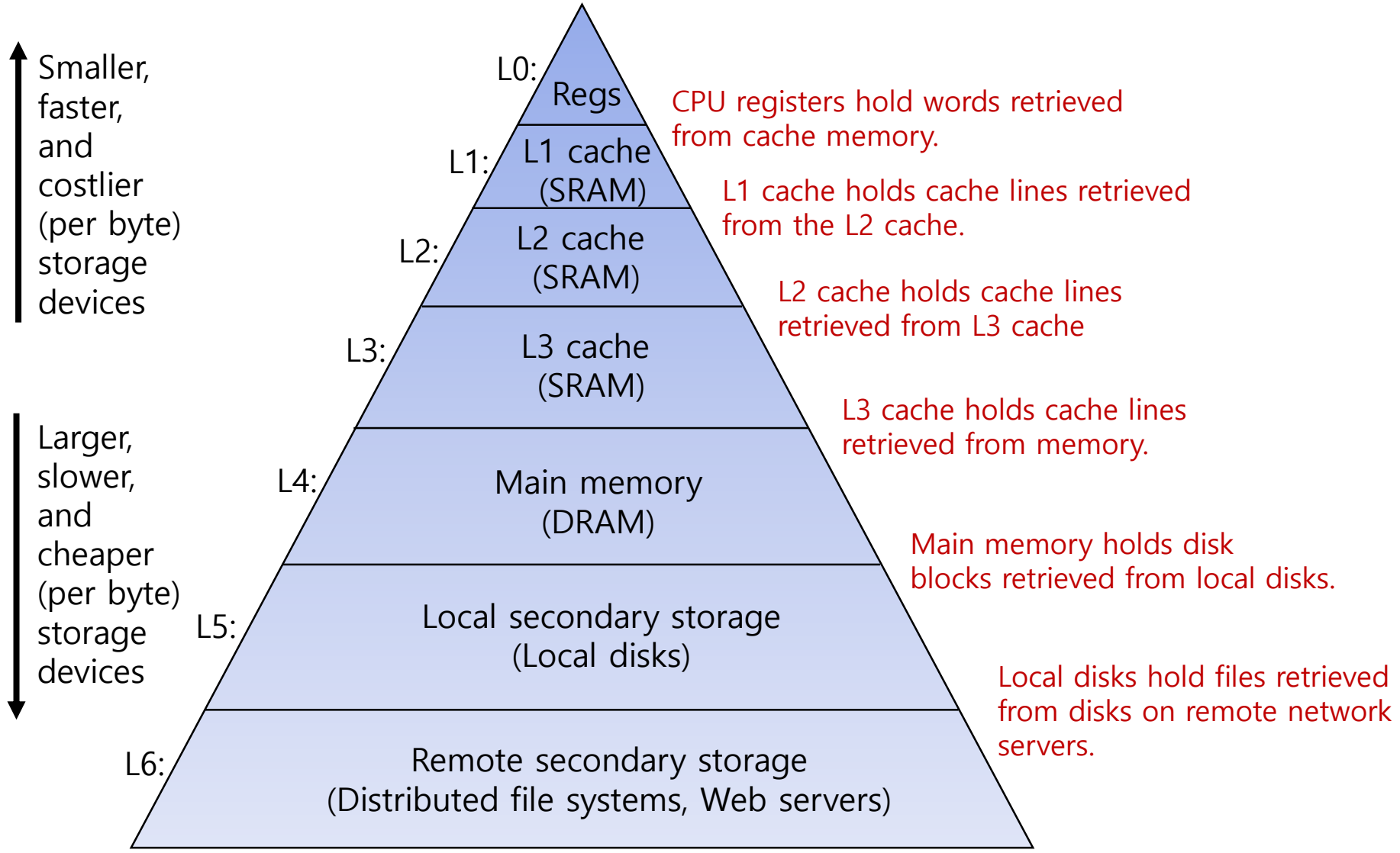
```
#define N 1000
typedef struct {
    int vel[3];
    int acc[3];
} point;

point p[N];
```

```
void clear1(point *p, int n) {
    int i, j;
    for (i = 0; i < n; i++) {
        for (j = 0; j < 3; j++)
            p[i].vel[j] = 0;
        for (j = 0; j < 3; j++)
            p[i].acc[j] = 0;
    }
}
```

```
void clear2(point *p, int n) {
    int i, j;
    for (i = 0; i < n; i++) {
        for (j = 0; j < 3; j++) {
            p[i].vel[j] = 0;
            p[i].acc[j] = 0;
        }
    }
}
```

```
void clear3(point *p, int n) {
    int i, j;
    for (j = 0; j < 3; i++) {
        for (i = 0; i < n; i++)
            p[i].vel[j] = 0;
        for (i = 0; i < n; i++)
            p[i].acc[j] = 0;
    }
}
```

Exploiting Memory Hierarchy



Contents

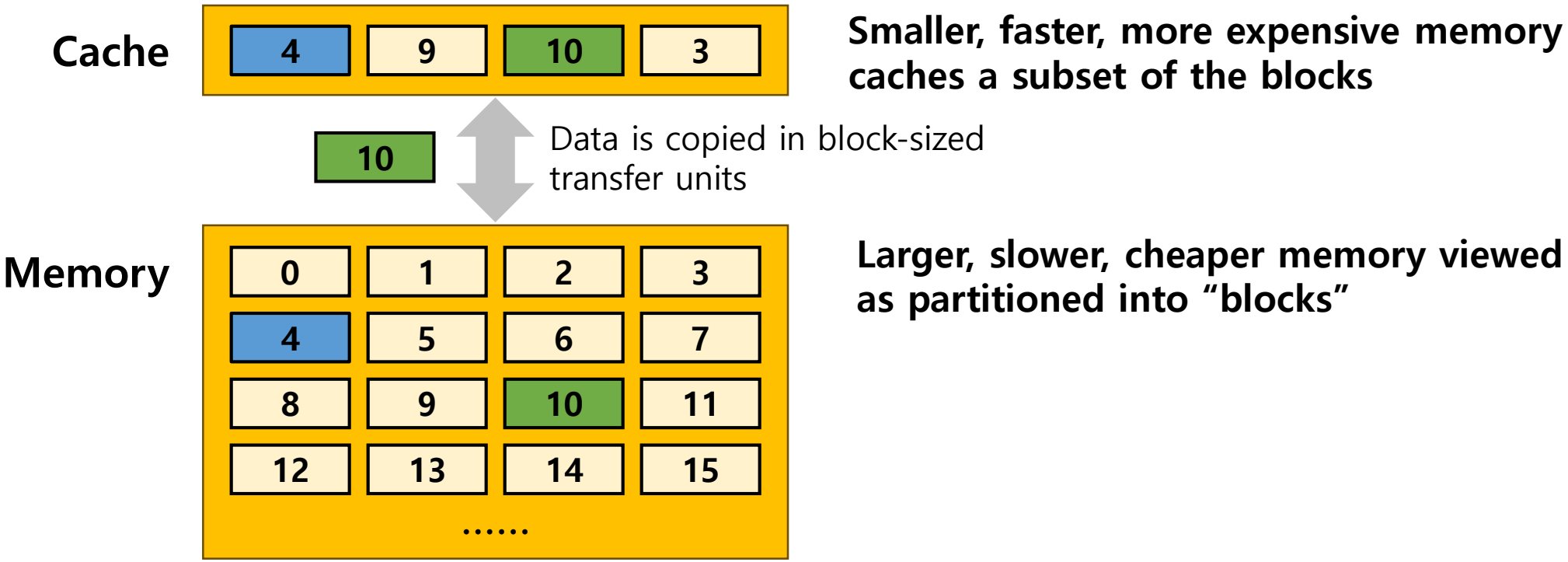
- The Memory Hierarchy
- Locality
- Caching
- Writing Cache-Friendly Code
- Cache Analysis and Valgrind

Caches

- A **cache** is a small, fast storage device on a CPU that holds limited subsets of main memory.

- Fundamental idea of a memory hierarchy:
  - For each k , the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1.
- Why do memory hierarchies work?
  - Because of locality, programs tend to access the data at level k more often than they access the data at level k+1.
  - Thus, the storage at level k+1 can be slower, and thus larger and cheaper per bit.

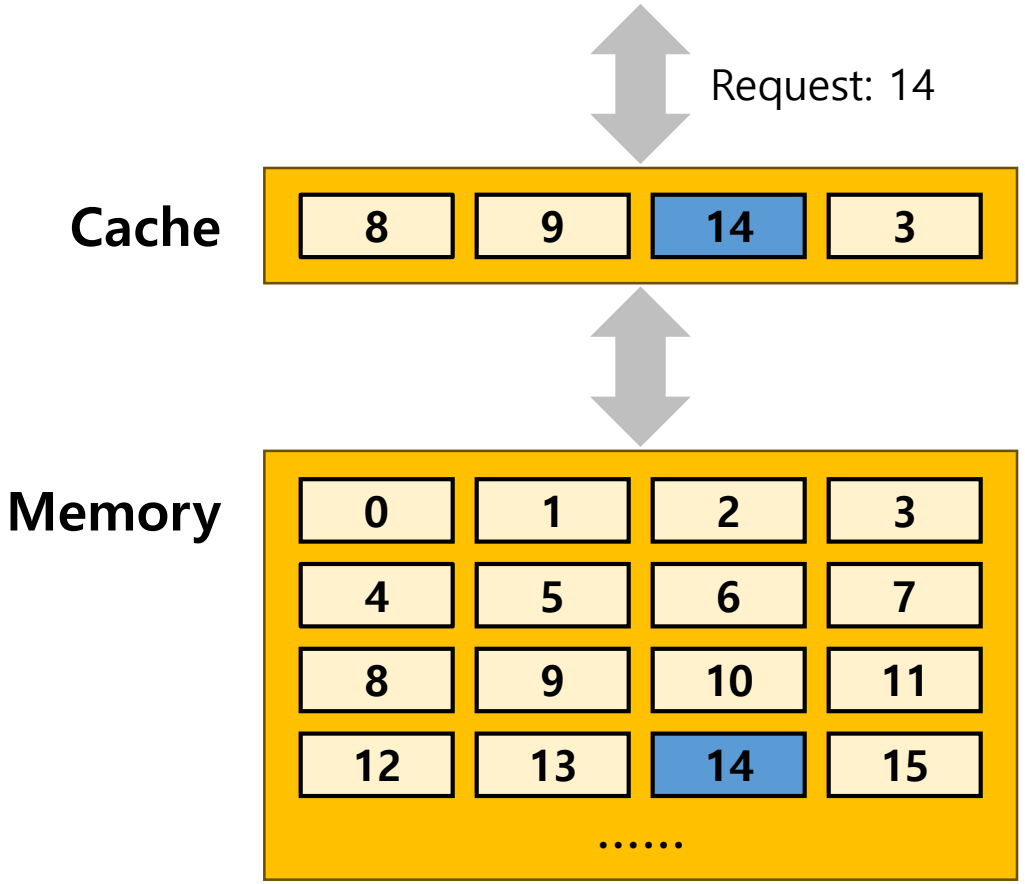
General Cache Concepts



Smaller, faster, more expensive memory caches a subset of the blocks

Larger, slower, cheaper memory viewed as partitioned into "blocks"

General Cache Concepts: Hit



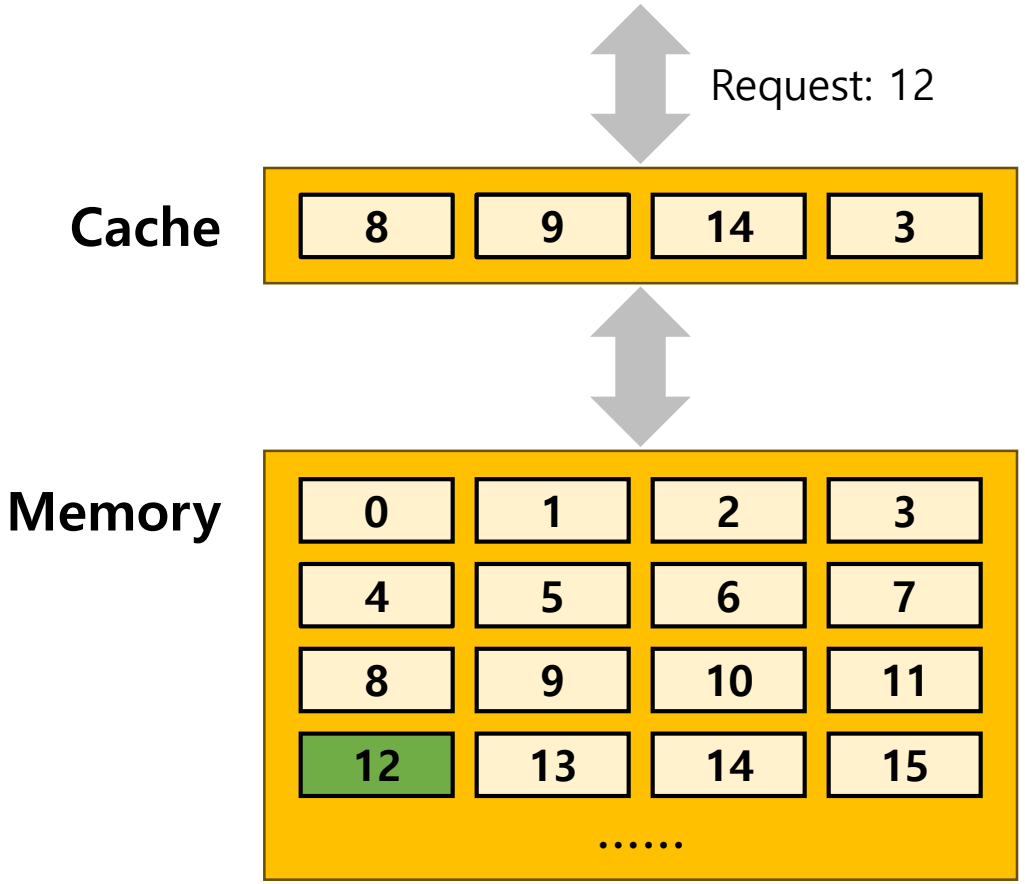
Data in block 14 is needed

Block 14 is in cache: Hit!

General Cache Concepts: Hit

- To maximize performance, the hardware simultaneously sends the desired address to both the cache and main memory.
- Because the cache is faster and closer to the ALU, the cache responds much more quickly than memory.
- If the data is present in the cache, the cache hardware cancels the pending memory access because the cache can serve the data faster than memory.

General Cache Concepts: Miss



Data in block 12 is needed

Block 12 is not in cache: Miss!

Block 12 is stored in cache

- Placement policy:
  determines where b goes
- Replacement policy:
  determines which block gets evicted (victim)

Impact of spatial locality on number of misses?

General Cache Concepts: Cache Miss

- If the data isn't in the cache, the CPU has no choice but to wait for memory to retrieve it.
- When the request to main memory completes, the CPU loads the retrieved data into the cache so that subsequent requests for the same address (which are likely thanks to temporal locality) can be serviced quickly from the cache.
- Even if the memory access that misses is writing to memory, the CPU still loads the value into the cache on a miss because it's likely that the program will attempt to access the same location again in the future.

General Cache Concepts: After Cache Miss

- When loading data into a cache after a miss, a CPU often finds that the cache doesn't have enough free space available.
- In such cases, the cache must first evict some resident data to make room for the new data that it's loading in.
- Because a cache stores subsets of data copied from main memory, evicting cached data that has been modified requires the cache to update the contents of main memory prior to evicting data from the cache.

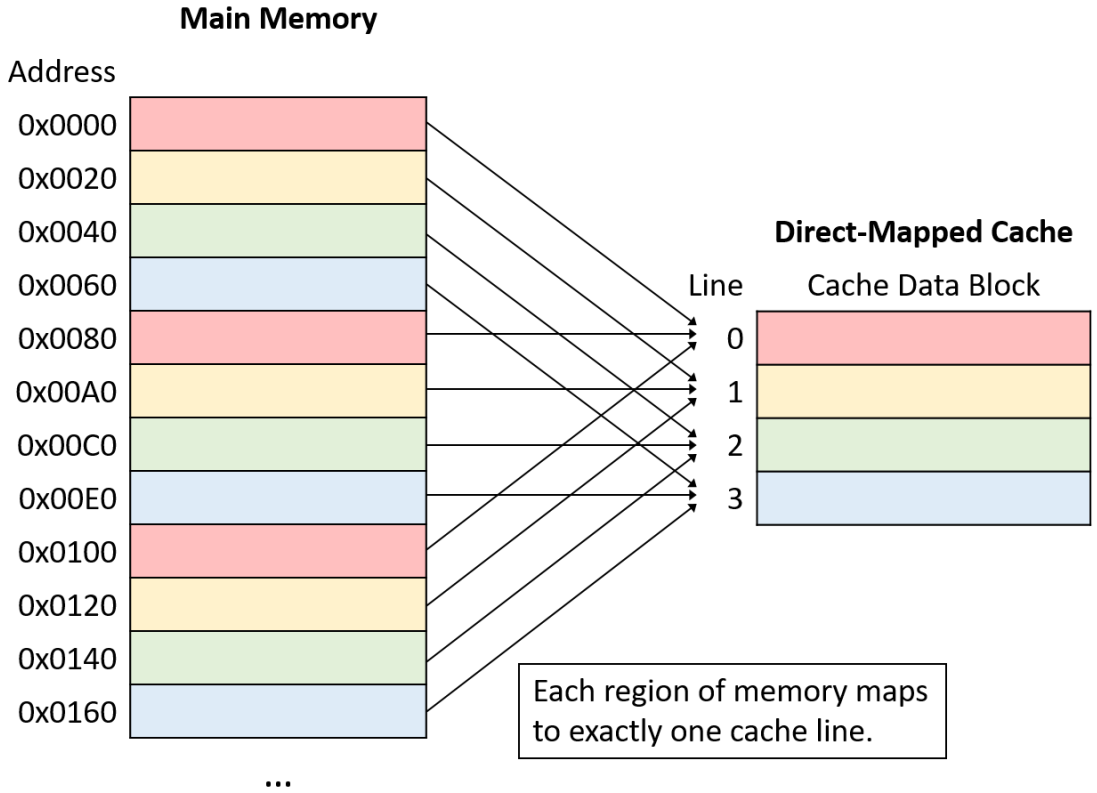
Design Questions for Caches

- *Which* subsets of a program's memory should the cache hold?
- *When* should the cache copy a subset of a program's data from main memory to the cache, or vice versa?
- *How* can a system determine whether a program's data is present in the cache?

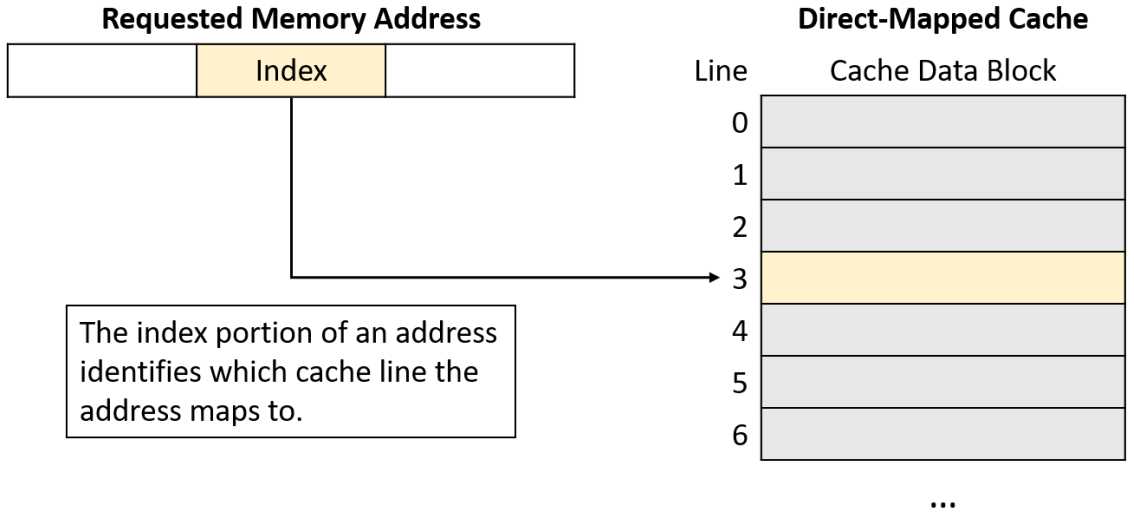
Direct-mapped Caches, Set-Associative Caches, and Fully Associative Caches

Direct-Mapped Caches

- Locating cached data



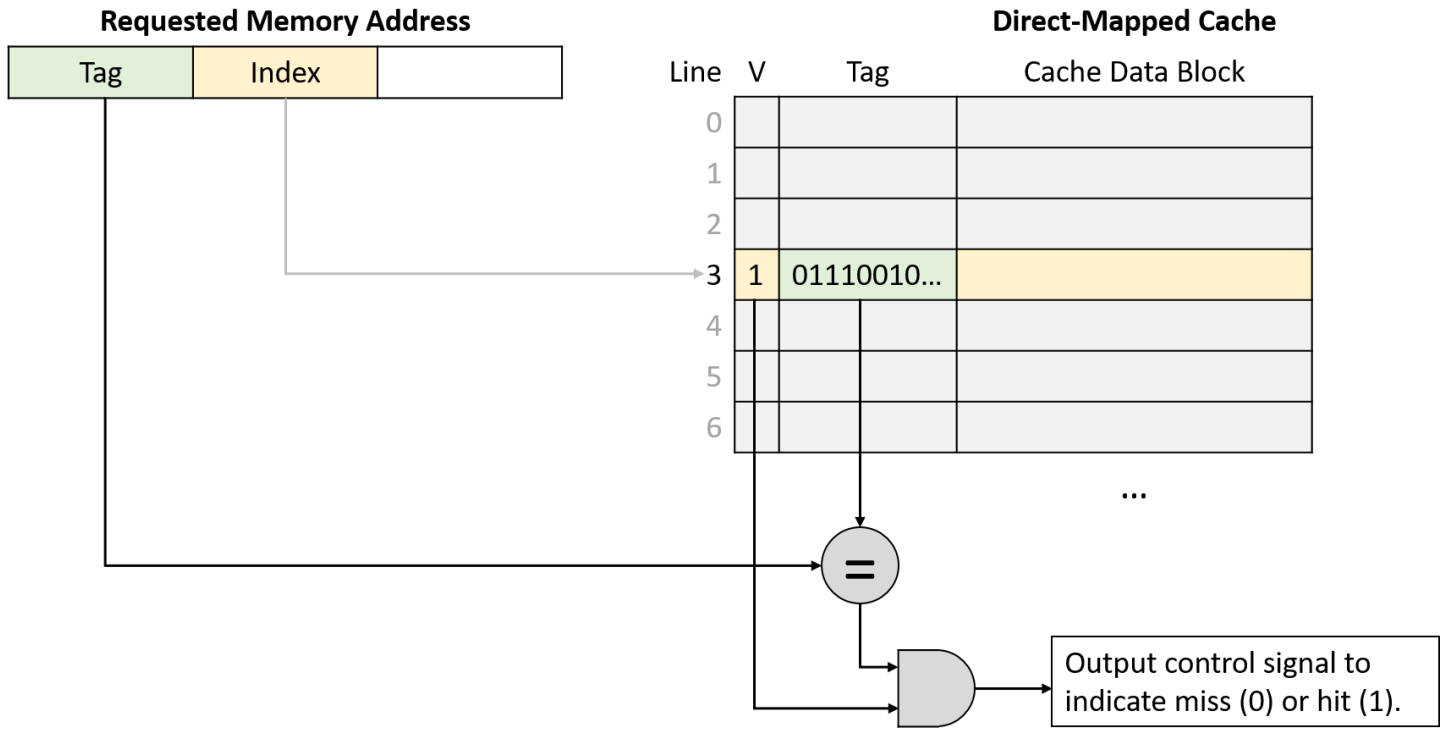
An example mapping of memory addresses to cache lines in a four-line direct-mapped cache with 32-byte cache blocks



The middle index portion of a memory address identifies a cache line.

Direct-Mapped Caches

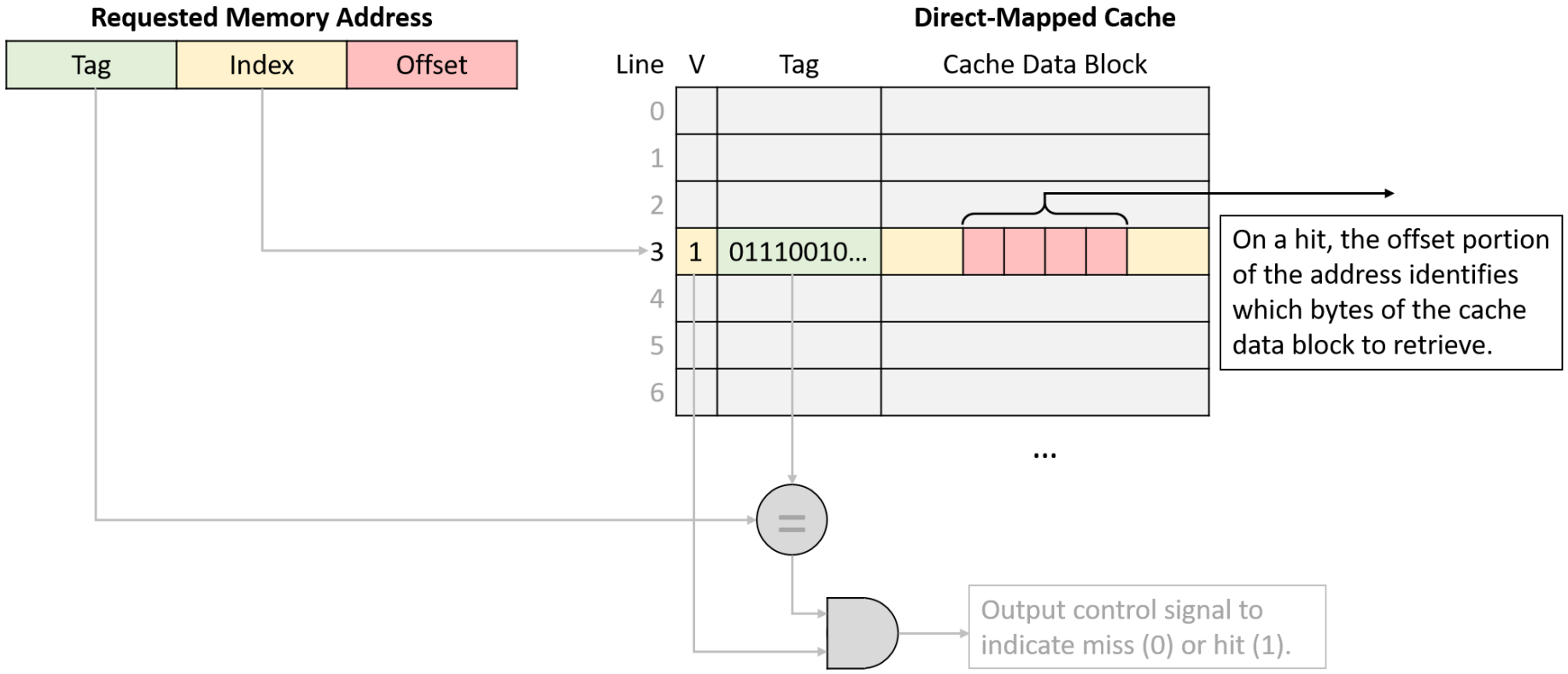
- Identifying Cache Contents



After using the requested memory address's index bits to locate the proper cache line, the cache simultaneously verifies the line's valid bit and checks its tag against the requested address's tag. If the line is valid with a matching tag, the lookup succeeds as a hit.

Direct-Mapped Caches

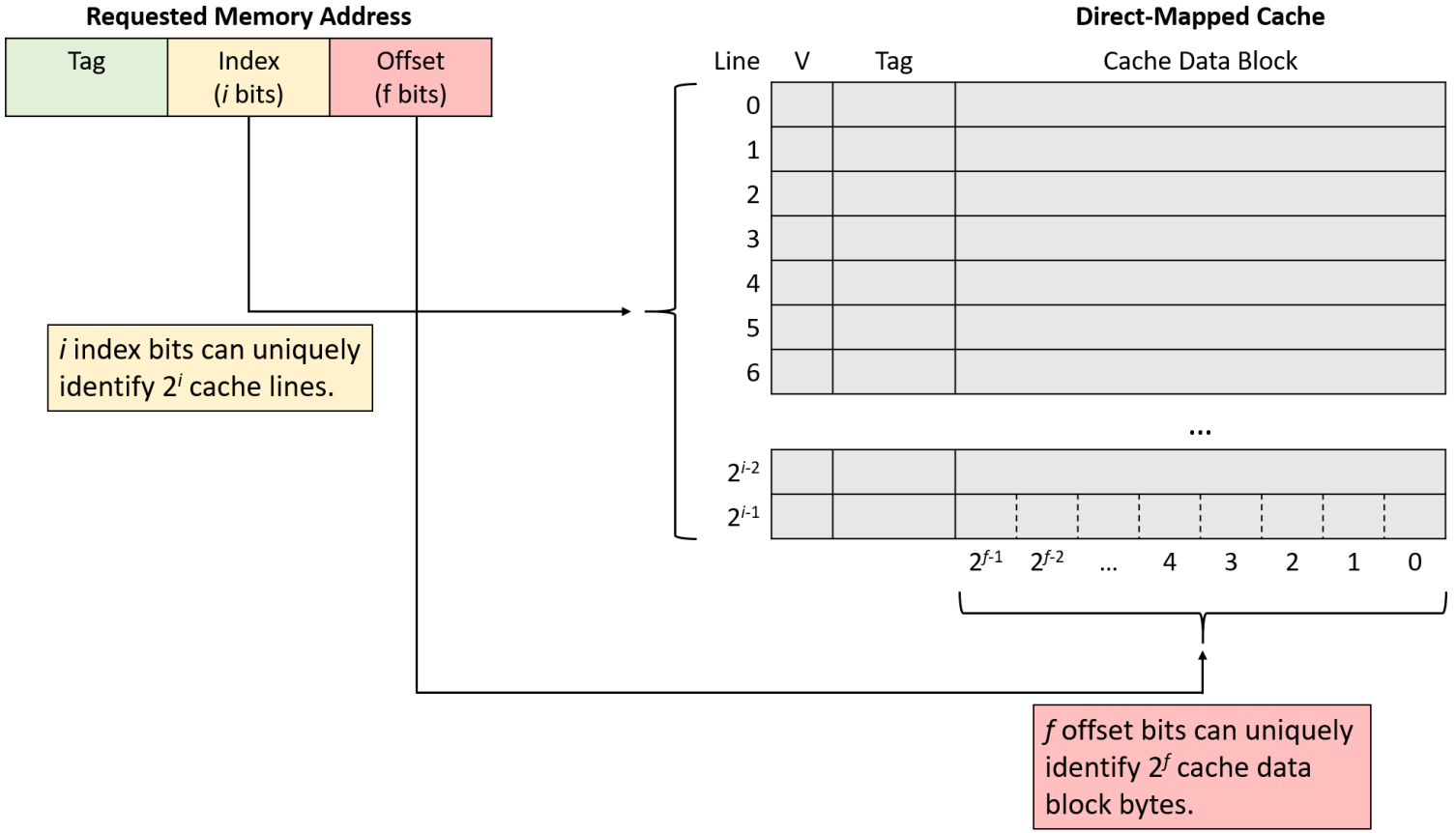
- Retrieving Cached Data



Given a cache data block, the offset portion of an address identifies which bytes the program wants to retrieve.

Direct-Mapped Caches

- Memory address division

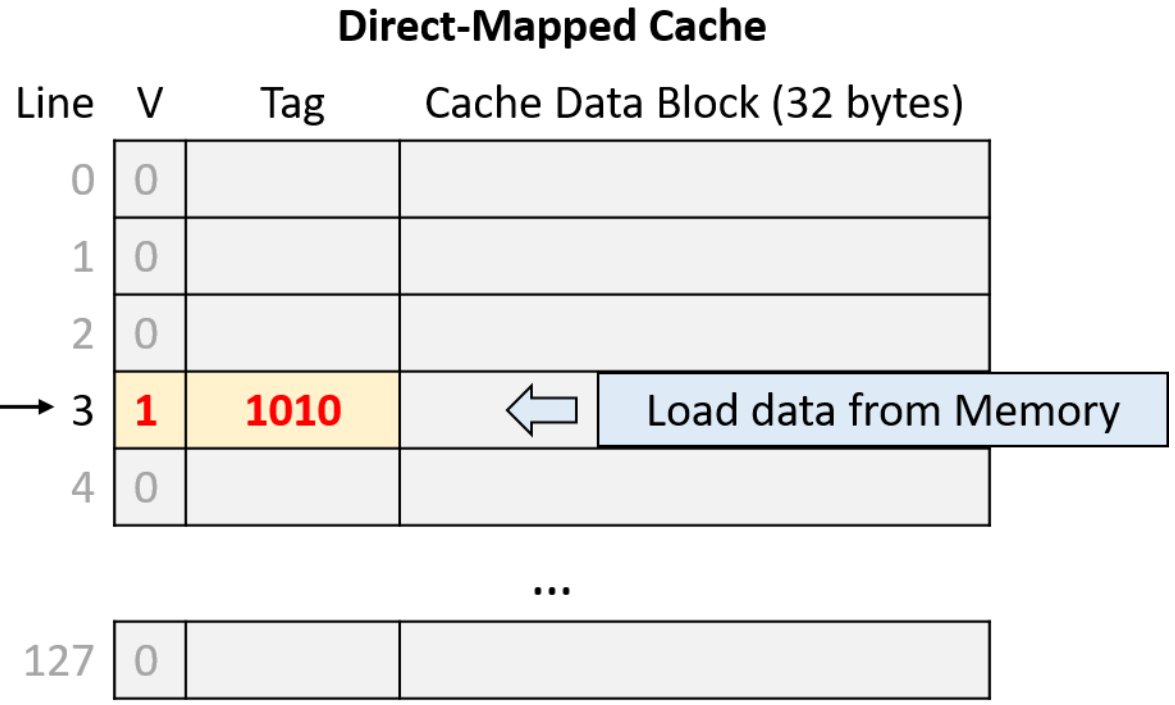


The index portion of an address uniquely identifies a cache line, and the offset portion uniquely identifies a position in the line's data block.

Direct-Mapped Caches: Example

Read from address 1010000001100100:

Tag	Index	Offset
1010	0000011	00100

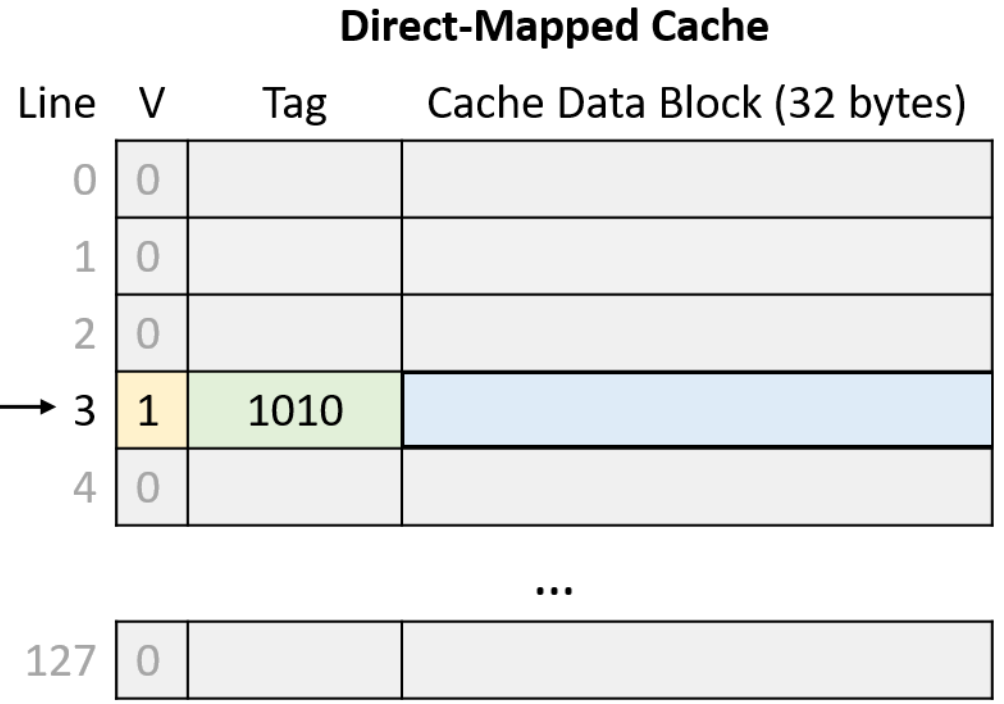


Result: miss, the line was invalid prior to this access.

Direct-Mapped Caches: Example

Read from address 1010000001100111:

Tag	Index	Offset
1010	0000011	00111

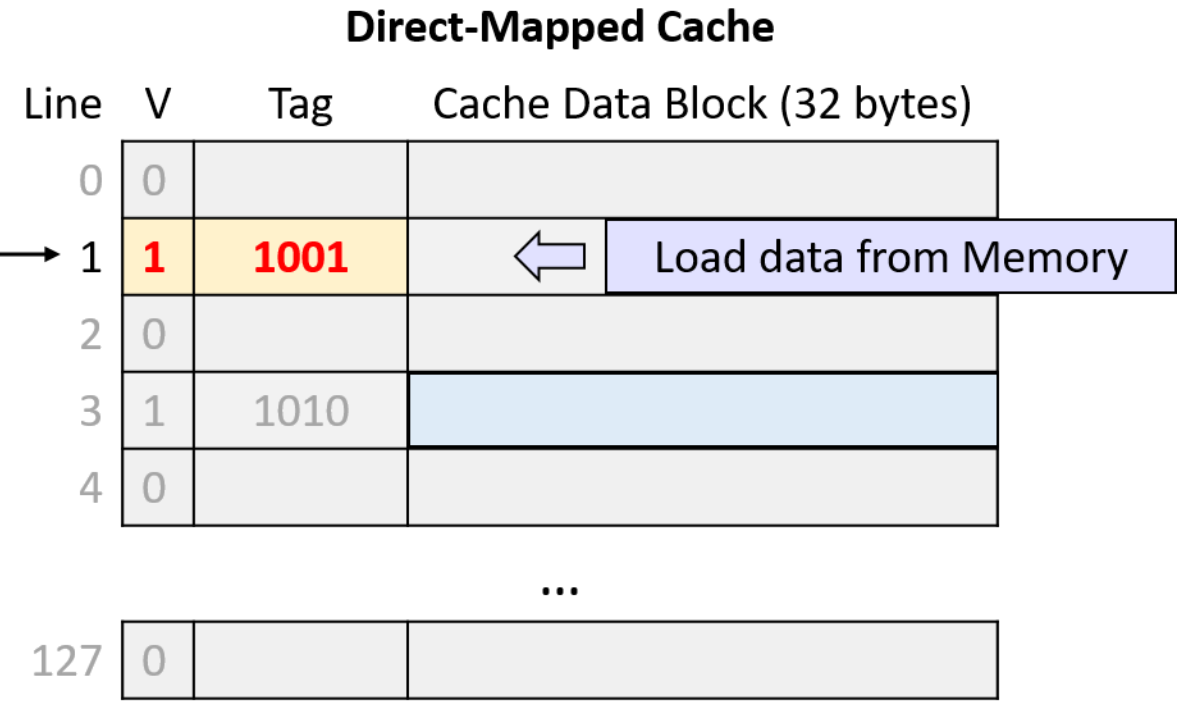


Result: hit, the line is valid, and the tag matches.

Direct-Mapped Caches: Example

Read from address 1001000000100000:

Tag	Index	Offset
1001	0000001	00000

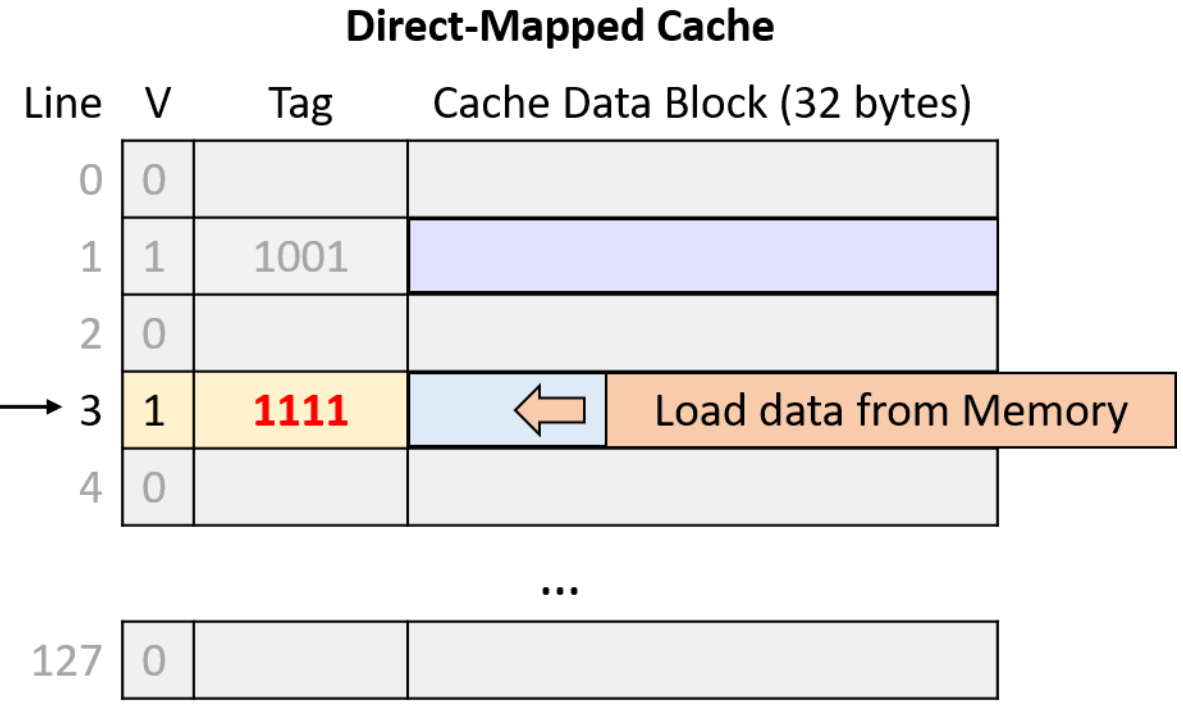


Result: miss, the line was invalid prior to this access.

Direct-Mapped Caches: Example

Read from address 1111000001100101:

Tag	Index	Offset
1111	0000011	00101



Result: miss, the line is valid, but the tag doesn't match.

Direct-Mapped Caches: Writing

- Write-through cache
  - A memory write operation modifies the value in the cache and simultaneously updates the contents of main memory.
  - That is, a write operation always synchronizes the contents of the cache and main memory immediately.

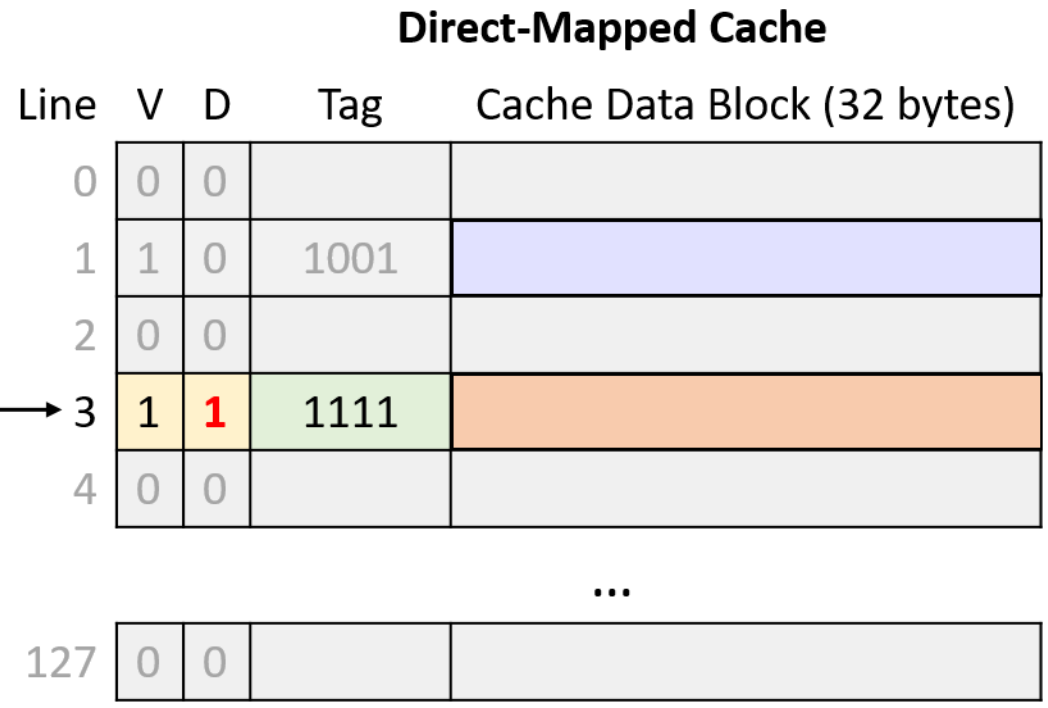
- Write-back cache
  - A memory write operation modifies the value stored in the cache's data block, but it does not update main memory.
  - Thus, after updating the cache's data, a write-back cache's contents differ from the corresponding data in main memory.

Direct-Mapped Caches: Example

- Write-Back

Write to address 1111000001100000:

Tag	Index	Offset
1111	0000011	00000



Result: hit, the line is valid, and the tag matches.

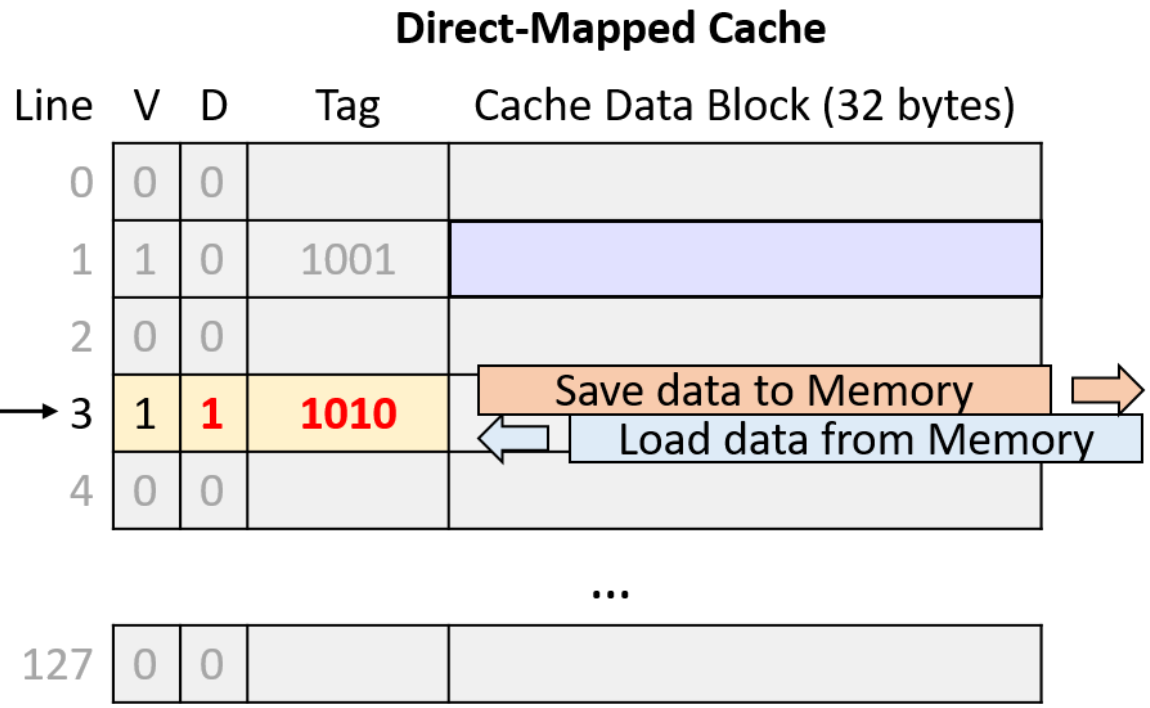
Set *dirty* bit to 1 on write.

Direct-Mapped Caches: Example

- Write-Back

Write to address 1010000001100100:

Tag	Index	Offset
1010	0000011	00100



Result: miss, the line is valid, but the tag doesn't match.

Save cache data block to memory before evicting it.

Set *dirty* bit to 1 on write. (again)

Types of Cache Misses

- Compulsory Misses (Cold-Start Misses)
  - Occur when a program accesses a memory location for the first time, with no prior data in the cache.
- Capacity Misses
  - Happen when the active data set exceeds the cache size, forcing the program to access memory locations not stored in the cache.
- Conflict Misses
  - Arise in certain cache designs, such as direct-mapped caches, where multiple memory addresses compete for the same cache line, leading to frequent evictions.
    - E.g., Referencing blocks 0, 8, 0, 8, … would miss every time

Contents

- The Memory Hierarchy
- Locality
- Caching
- Writing Cache-Friendly Code
- Cache Analysis and Valgrind

Basic Approach for Cache Friendly Code

- Make the common case go fast.
  - Programs often spend most of their time in a few core function. These functions often spend most of their time in a few loop. So focus on the inner loops of the core functions and ignore the rest.

- Minimize the number of cache misses in each inner loop.
  - All other things being equal, such as the total number of loads and stores, loops with better miss rates will run faster.

Cache-Friendly Function

- Suppose that v is block aligned, words are 4bytes, cache blocks are 4 words, and the cache is initially empty.

Is this function cache friendly?

```
int sumvec(int v[N]) {
    int i, sum = 0;

    for (i = 0; i < N; i++)
        sum += v[i];

    return sum
}
```

In general, if a cache has a block size of `B` bytes, then a stride-k reference pattern (where k is expressed in words) results in an average of `min(1, (wordsize*k)/B)` misses per loop iteration.

v[i]	i=0	i=1	i=2	i=3	i=4	i=5	i=6	i=7
Access order, [h]it or [m]iss	1 [m]	2 [h]	3 [h]	4 [h]	5 [m]	6 [h]	7 [h]	8 [h]

Cache-Friendly Function

- 2D-Array sum in row-major order

```
int sumarrayrows(int a[M][N]) {
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];

    return sum;
}
```

a[i][j]	j=0	j=1	j=2	j=3	j=4	j=5	j=6	j=7
i=0	1 [m]	2 [h]	3 [h]	4 [h]	5 [m]	6 [h]	7 [h]	8 [h]
i=1	9 [m]	10 [h]	11 [h]	12 [h]	13 [m]	14 [h]	15 [h]	16 [h]
i=2	17 [m]	18 [h]	19 [h]	20 [h]	21 [m]	22 [h]	23 [h]	24 [h]
i=3	25 [m]	26 [h]	27 [h]	28 [h]	29 [m]	30 [h]	31 [h]	32 [h]

Cache-Friendly Function

- 2D-Array sum in column-major order

```
int sumarraycols(int a[M][N]) {
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; j < M; i++)
            sum += a[i][j];

    return sum;
}
```

a[i][j]	j=0	j=1	j=2	j=3	j=4	j=5	j=6	j=7
i=0								
i=1								
i=2								
i=3								

Cache-Friendly Function: Example #1

- Determine the cache performance of the following code on a machine with a 2048-byte direct-mapped data cache with 32-byte blocks (B=32).
  - `sizeof(int)=4` / `grid` begins at memory address 0 / Cache is initially empty.
  - The only memory accesses are to the entries of the array `grid`.

```
struct algae_position {
    int x;
    int y;
};

struct algae_position grid[32][32]
int total_x = 0, total_y = 0;
int i, j;
```

```
for (i = 31; i >= 0; i--)
    for (j = 31; j >= 0; j--)
        total_x += grid[i][j].x;

for (i = 31; i >= 0; i--)
    for (j = 31; j >= 0; j--)
        total_y += grid[i][j].y;
```

  - What is the total number of reads?
  - What is the total number of reads that miss in the cache?
  - What is the miss rate?

Cache-Friendly Function: Example #2

- Given the assumptions of the previous slide, determine the cache performance of the following code:

```
for (i = 31; i >= 0; i--) {
    for (j = 31; j >= 0; j--) {
        total_x += grid[j][i].x;
        total_y += grid[j][i].y;
    }
}
```

  - What is the total number of reads?
  - What is the total number of reads that hit in the cache?
  - What is the hit rate?
  - What would the miss hit be if the cache were twice as big?

Cache-Friendly Function: Example #3

- Given the assumptions of the previous slide, determine the cache performance of the following code:

```
for (i = 31; i >= 0; i--) {
    for (j = 31; j >= 0; j--) {
        total_x += grid[i][j].x;
        total_y += grid[i][j].y;
    }
}
```

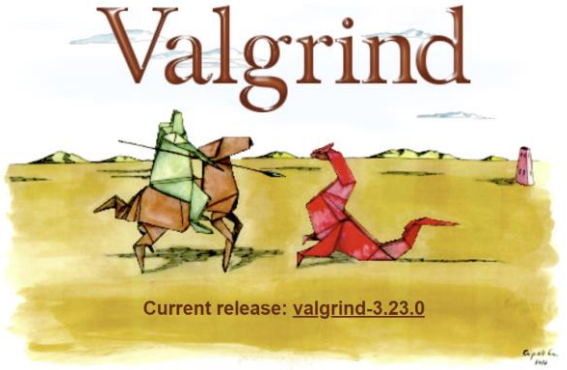
  - What is the total number of reads?
  - What is the total number of reads that hit in the cache?
  - What is the hit rate?
  - What would the hit rate be if the cache were twice as big?

Contents

- The Memory Hierarchy
- Locality
- Caching
- Writing Cache-Friendly Code
- Cache Analysis and Valgrind

Valgrind and Cachegrind

- Valgrind is a programming tool for memory debugging, memory leak detection, and profiling.
  - Memory error detection: Identifies invalid memory accesses, memory leaks, and the use of uninitialized value
  - Runtime analysis of the program: Tracks memory usage and allocation status.
- Cachegrind is a cache profiler.
  - Tracks cache misses and branch prediction failures.
  - Analyzes L1 and L2 cache read/write performance to identify potential optimization opportunities.
  - Visualizes cache performance issues through output data and suggests directions for code optimization.



Theoretical Analysis and Benchmarking

- Row-major order vs. Column-major order (cachex.c)

```c
float averageMat_v1(int **mat, int n) {
    int i, j, total = 0;

    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            // Note indexing: [i][j]
            total += mat[i][j];
        }
    }

    return (float) total / (n * n);
}
```

```c
float averageMat_v2(int **mat, int n) {
    int i, j, total = 0;

    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            // Note indexing: [j][i]
            total += mat[j][i];
        }
    }

    return (float) total / (n * n);
}
```

Theoretical Analysis and Benchmarking

- Row-major order vs. Column-major order (cachex.c)

```
int main(int argc, char** argv) {
    ......
    /* Time version 1. */
    gettimeofday(&tstart, NULL);
    res = averageMat_v1(matrix, n);
    gettimeofday(&tend, NULL);
    timer = tend.tv_sec - tstart.tv_sec + (tend.tv_usec - tstart.tv_usec)/1.e6;
    printf("v1 average is: %.2f; time is %g\n", res, timer);

    /* Time version 2. */
    gettimeofday(&tstart, NULL);
    res = averageMat_v2(matrix, n);
    gettimeofday(&tend, NULL);
    timer = tend.tv_sec - tstart.tv_sec + (tend.tv_usec - tstart.tv_usec)/1.e6;
    printf("v2 average is: %.2f; time is %g\n", res, timer);
```

```
yunmin@peace:~/ch8$ gcc cachex.c -o cachex
yunmin@peace:~/ch8$ ./cachex 5000
v1 average is: 50.49; time is 0.06231
v2 average is: 50.49; time is 0.247689
```

Cache Analysis using Cachegrind

- Cachegrind collects and outputs the following information:
  - `Ir`: Instruction cache reads
  - `I1mr`: `L1` instruction cache read misses
  - `ILmr`: `LL` cache instruction read misses
  - `Dr`: Data cache reads
  - `D1mr`: `D1` cache read misses
  - `DLmr`: `LL` cache data misses
  - `Dw`: Data cache writes
  - `D1mw`: `D1` cache write misses
  - `DLmw`: `LL` cache data write misses
  - `D1` total access is computed by `D1 = D1mr + D1mw`
  - `LL` total access is given by `ILmr + DLmr + DLmw`

* LL Cache: Last-Level Cache (L3 Cache in the peace server) https://valgrind.org/docs/manual/cg-manual.html

Cache Analysis using Cachegrind

```
yunmin@peace:~/ch8$ valgrind --tool=cachegrind ./cachex 1000
==9100== Cachegrind, a cache and branch-prediction profiler
==9100== Copyright (C) 2002-2015, and GNU GPL'd, by Nicholas Nethercote et al.
==9100== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==9100== Command: ./cachex 1000
==9100==
--9100-- warning: L3 cache found, using its data for the LL simulation.
--9100-- warning: specified LL cache: line_size 64  assoc 20  total_size 26,214,400
--9100-- warning: simulated LL cache: line_size 64  assoc 25  total_size 26,214,400
v1 average is: 50.52; time is 0.061665
v2 average is: 50.52; time is 0.077591
==9100==
==9100== I   refs:      100,549,469
==9100== I1  misses:          1,147
==9100== LLi misses:          1,129
==9100== I1  miss rate:        0.00%
==9100== LLi miss rate:        0.00%
==9100==
==9100== D   refs:       50,225,588  (42,152,155 rd   + 8,073,433 wr)
==9100== D1  misses:      1,256,579  ( 1,192,918 rd   +    63,661 wr)
==9100== LLd misses:         65,365  (     1,973 rd   +    63,392 wr)
==9100== D1  miss rate:         2.5% (       2.8%     +       0.8%  )
==9100== LLd miss rate:         0.1% (       0.0%     +       0.8%  )
==9100==
==9100== LL refs:         1,257,726  ( 1,194,065 rd   +    63,661 wr)
==9100== LL misses:          66,494  (     3,102 rd   +    63,392 wr)
==9100== LL miss rate:          0.0% (       0.0%     +       0.8%  )
yunmin@peace:~/ch8$ ls -l cachegrind.out.*
-rw------- 1 yunmin yunmin 89041 10월 21 01:03 cachegrind.out.9100
```

We are interested in the hits and misses for the two versions of this averaging function, so we should use Cachgrind tool cg_annotate.

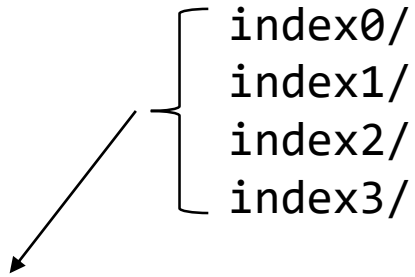
Cachgrind produces this file, and we can read it by running cg_annotate.

Cache Analysis using Cachegrind

```
yunmin@peace:~/ch8$ cg_annotate cachegrind.out.8482
--------------------------------------------------------------------------------
I1 cache:         32768 B, 64 B, 8-way associative
D1 cache:         32768 B, 64 B, 8-way associative
LL cache:         26214400 B, 64 B, 25-way associative
Command:          ./cachex 1000
Data file:        cachegrind.out.8482
Events recorded:  Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Events shown:     Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Event sort order: Ir I1mr ILmr Dr D1mr DLmr Dw D1mw DLmw
Thresholds:       0.1 100 100 100 100 100 100 100 100
Include dirs:
User annotated:
Auto-annotation:  off

--------------------------------------------------------------------------------
         Ir  I1mr  ILmr         Dr      D1mr  DLmr        Dw   D1mw   DLmw
--------------------------------------------------------------------------------
100,549,477 1,149 1,131 42,152,159 1,192,918 1,973 8,073,427 63,661 63,392  PROGRAM TOTALS

--------------------------------------------------------------------------------
        Ir I1mr ILmr        Dr      D1mr DLmr        Dw   D1mw   DLmw  file:function
--------------------------------------------------------------------------------
25,975,792    2    2 8,002,480         4    0 4,001,240      0      0  /build/glibc-S7Ft5T/glibc-2.23/stdlib/random_r.c:random_r
20,020,022    4    4 8,008,007         1    0 2,003,007 61,875 61,874  ???:genRandomMatrix
17,000,000    3    3 6,000,000         1    0 1,000,000      0      0  /build/glibc-S7Ft5T/glibc-2.23/stdlib/random.c:random
16,009,019    3    3 9,005,007    62,877    0     1,005      0      0  ???:averageMat_v1
16,009,019    1    1 9,005,007 1,125,993    0     1,005      0      0  ???:averageMat_v2
 4,000,000    1    1 1,000,000         0    0 1,000,000      0      0  /build/glibc-S7Ft5T/glibc-2.23/stdlib/rand.c:rand
 1,002,222   26   18 1,002,122         9    0        34      1      1  ???:???
   135,953   21   21    40,966     1,007    0    18,016      0      0  /build/glibc-S7Ft5T/glibc-2.23/malloc/malloc.c:_int_free
   135,096   19   19    17,065         7    1    24,991    975    972  /build/glibc-S7Ft5T/glibc-2.23/malloc/malloc.c:_int_malloc
```

Processor and Cache Information on Linux

- `$ lscpu`
- `$ cat /proc/cpuinfo`



- `$ cat /sys/devices/system/cpu/cpu0/cache/index*/type`
- `$ cat /sys/devices/system/cpu/cpu0/cache/index*/level`
- `$ cat /sys/devices/system/cpu/cpu0/cache/index*/shared_cpu_list`