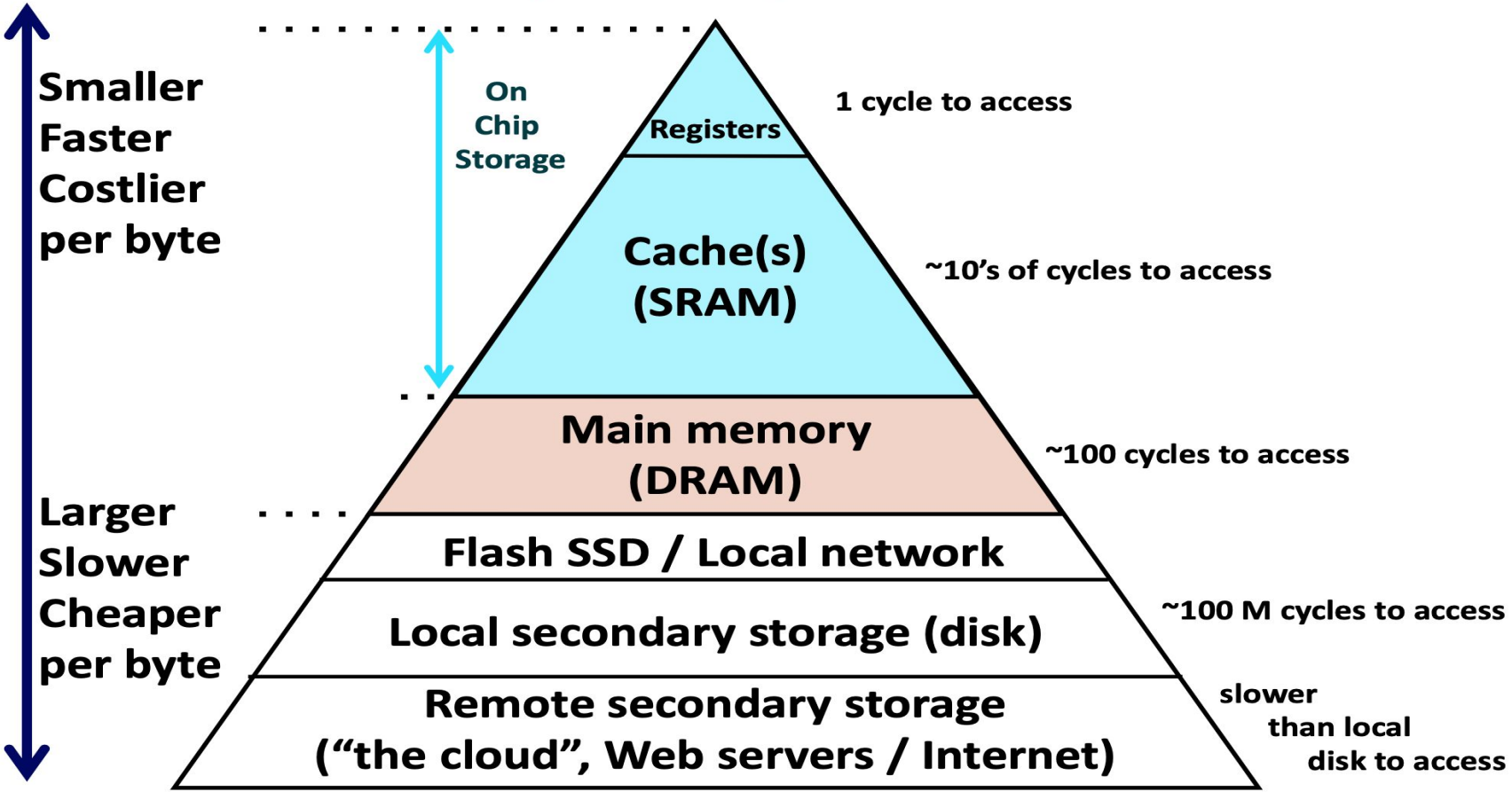
Memory organization

COMP201 Lab Session
Fall 2020

KOÇ
UNIVERSITY

The Memory Hierarchy



Why do we need Memory Hierarchies?

Some fundamental properties of computer system

- Fast storage technologies cost more per byte, have less capacity, and require more power (heat!).
- The gap between CPU and main memory speed is widening.
- Well-written programs tend to exhibit good locality.

These fundamental properties of hardware and software suggest an approach for organizing memory and storage systems known as a memory hierarchy.

Fundamental idea of a memory hierarchy

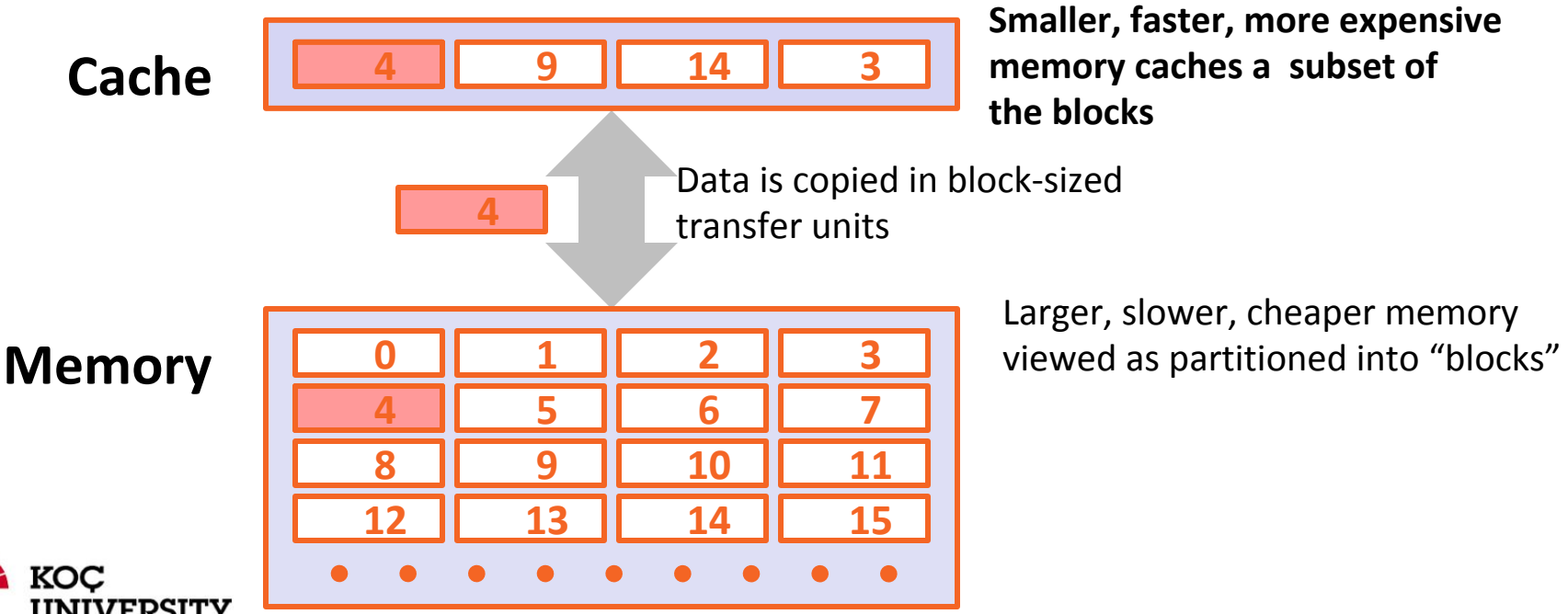
- For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1.
- Because of locality, programs tend to access the data at level k more
- often than they access the data at level k+1.

(Ideal): The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.

Examples of Caching in the Mem. Hierarchy

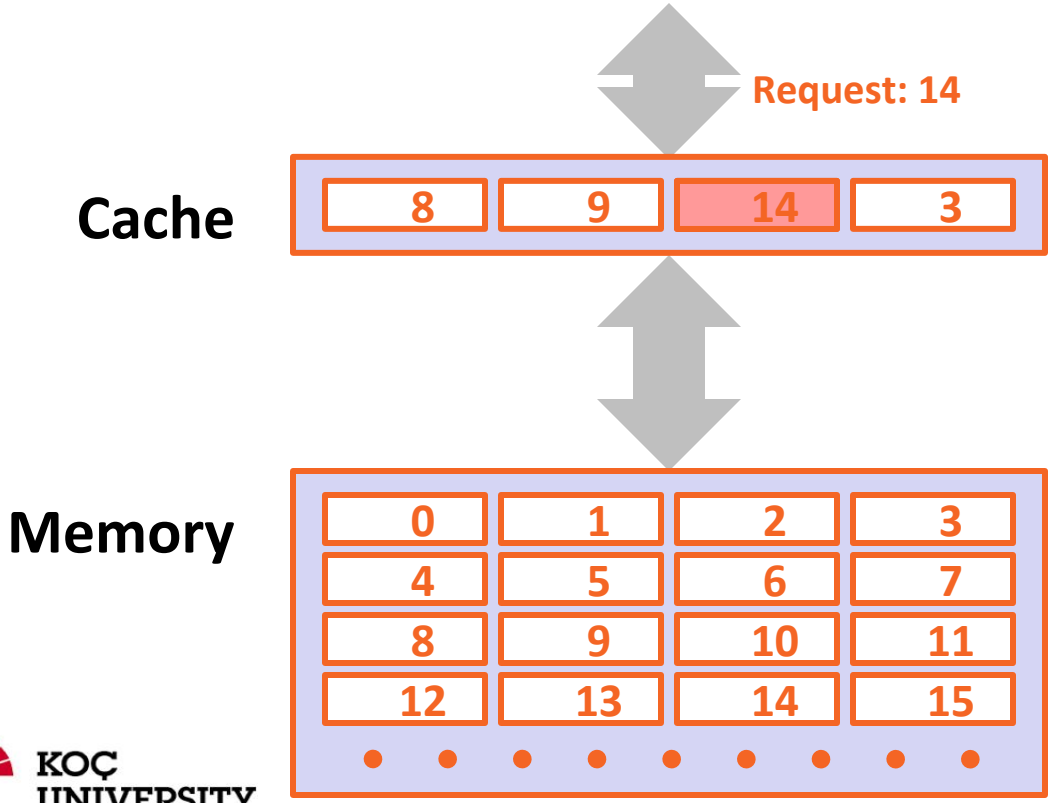
Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-8 bytes words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware MMU
L1 cache	64-byte blocks	On-Chip L1	4	Hardware
L2 cache	64-byte blocks	On-Chip L2	10	Hardware
Virtual Memory	4-KB pages	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Disk firmware
Network buffer cache	Parts of files	Local disk	10,000,000	NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

Recall: General Cache Concepts



KOÇ UNIVERSITY

General Cache Concepts: Hit

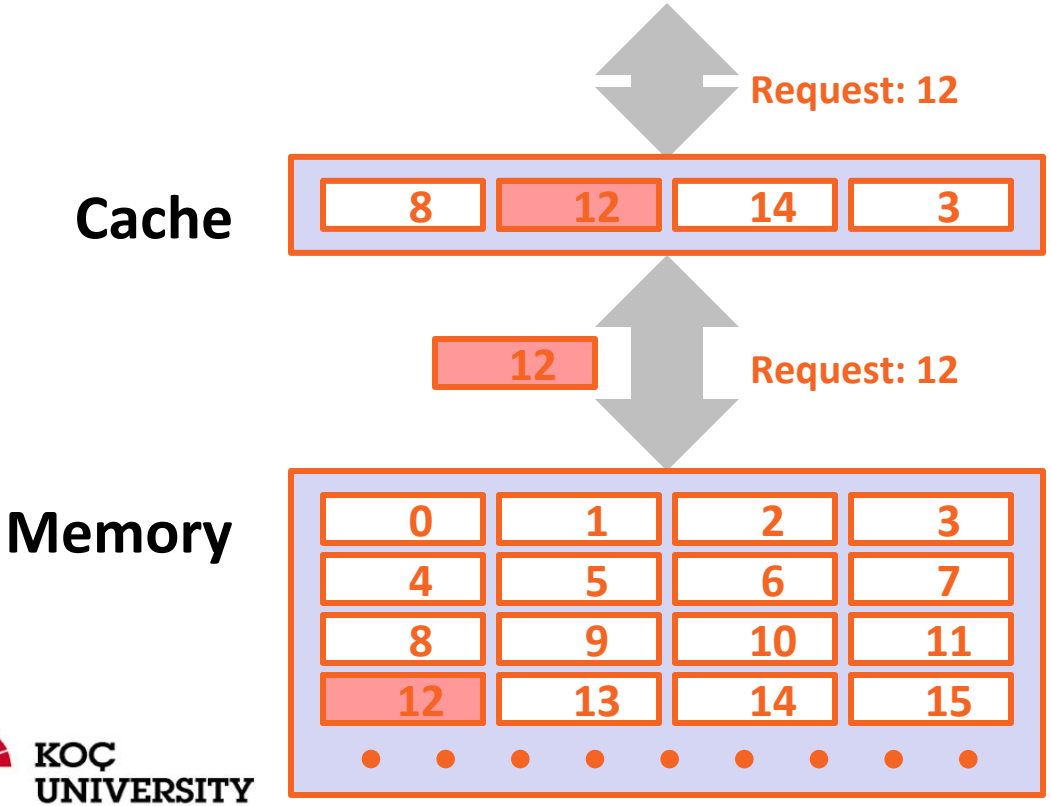


Data in block 14 is needed

Block 14 is in cache:
Hit!

KOÇ UNIVERSITY

General Cache Concepts: Miss



Data in block 12 is needed

Block 12 is not in cache:
Miss!

Block 12 is fetched from memory

Block 12 is stored in cache

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

KOÇ UNIVERSITY

Recall: General Caching Concepts: 3 Types of Cache Misses

- Cold (compulsory) miss
  - Cold misses occur because the cache starts empty and this is the first reference to the block.
- Capacity miss
  - Occurs when the set of active cache blocks (working set) is larger than the cache.
- Conflict miss
  - Most catches limit blocks at level k+1 to a small subset (sometimes a singleton) of the block positions at level k.
    - E.g. Block i at level k+1 must be placed in block (i mod 4) at level k.
  - Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block.
    - E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.

Principle of Locality

Principle of Locality:

- Programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves.
- Temporal locality:
  - Recently referenced items are likely to be referenced in the near future.
- Spatial locality:
  - Items with nearby addresses tend to be referenced close together in time.

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

Locality Example:

- Data
  - Reference array elements in succession(stride-1 reference pattern): Spatial locality
  - Reference sum each iteration: Temporal locality
- Instructions
  - Reference instructions in sequence: Spatial locality
  - Cycle through loop repeatedly: Temporal locality

Locality Example

Does this function have a good locality?
How we can improve it?

Hint: C arrays are allocated in row-major

```
int sumarray3d(int a[M][N][N])
{
    int i, j, k, sum = 0;

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

Concluding Observations

Programmer can optimize for cache performance

- How data structures are organized
- How data are accessed
  - Nested loop structure
  - Blocking is a general technique

All systems favor “cache friendly code”

- Getting absolute optimum performance is very platform specific
  - Cache sizes, line sizes, associatives, etc.
- Can get most of the advantage with generic code
  - Keep working set reasonably small (temporal locality)
  - Use small strides (spatial locality)

KOÇ
UNIVERSITY

Callgrind

Code Profiling

- A **code profiler** is a tool to analyze a program and report on its resource usage
  - "resource" could be memory, CPU cycles, network bandwidth, and so on
- The program is run under control of a profiling tool
- During application development, a common step is to improve runtime performance using profiling tools.
- To not waste time on optimizing functions which are rarely used, one needs to know in which parts of the program most of the time is spent.
- Some example:
  - Callgrind, GProf, JConsol, CLR

Valgrind

the Valgrind framework supports a variety of runtime analysis tools

- memcheck
  - detects memory errors/leaks
- massif
  - reports on heap usage
- helgrind
  - detects multithreaded race conditions
- callgrind/cachegrind
  - profiles CPU/cache performance

KOÇ
UNIVERSITY

Callgrind/cachegrind

- The Valgrind profiling tools are cachegrind and callgrind
- The cachegrind tool simulates the L1/L2 caches and counts cache misses/hits.
- The callgrind tool counts function calls and the CPU instructions executed within each call and builds a function callgraph
- The callgrind tool includes a cache simulation feature adopted from cachegrind, so you can actually use callgrind for both CPU and cache profiling.

Basic Usage of Callgrind

- First, we need to compile our program with debugging enabled
  - gcc -g -ggdb name.c -o name.out
- You first need to run your program under Valgrind and explicitly request the callgrind tool (if unspecified, the tool defaults to memcheck)
  - valgrind --tool=callgrind [possible options] name.out program-arguments
- The result will be stored on the files callgrind.out.PID, where PID will be the process identifier.

Process identifier

```
==22417== Events    : Ir
==22417== Collected : 7247606
==22417==
==22417== I   refs:      7,247,606
```

Number of Instruction read (Ir)

Basic Usage of Callgrind

Counting instructions with callgrind

- The callgrind output file is a text file, but its contents are not intended for you to read yourself.
- You can properly read the output using callgrind_annotate
  - callgrind_annotate --auto=yes callgrind.out.PID
- The --auto=yes option report counts for each C statement
- Do not forget to replace PID by the actual number.

This program sort a 1000-member array using selection sort

```
    . void swap(int *a, int *b)
    3,000  {
    3,000      int tmp = *a;
    4,000      *a = *b;
    3,000      *b = tmp;
    2,000  }
        .
        . int find_min(int arr[], int start, int stop)
    3,000  {
    2,000      int min = start;
2,005,000      for(int i = start+1; i <= stop; i++)
4,995,000          if (arr[i] < arr[min])
    6,178              min = i;
    1,000      return min;
    2,000  }
        . void selection_sort(int arr[], int n)
        3  {
    4,005      for (int i = 0; i < n; i++) {
    9,000          int min = find_min(arr, i, n-1);
7,014,178  => sorts.c:find_min (1000x)
   10,000          swap(&arr[i], &arr[min]);
   15,000  => sorts.c:swap (1000x)
        .    }
        2  }
        .
```

KOÇ UNIVERSITY

Basic Usage of Callgrind

Interpreting the results

- The Ir counts are basically the count of assembly instructions executed.
- By default, the counts are *exclusive*
  - The counts for a function include only the time spent in that function and not in the functions that it calls.
- By using exclusive counts you can detect the bottlenecks.
- For example, in the right code, the work is concentrated in the loop to find the min value
  - Conclusion: Caching the min array element is useful here.

```
    . void swap(int *a, int *b)
    3,000  {
    3,000      int tmp = *a;
    4,000      *a = *b;
    3,000      *b = tmp;
    2,000  }
        .
        . int find_min(int arr[], int start, int stop)
    3,000  {
    2,000      int min = start;
2,005,000      for(int i = start+1; i <= stop; i++)
4,995,000          if (arr[i] < arr[min])
    6,178              min = i;
    1,000      return min;
    2,000  }
        . void selection_sort(int arr[], int n)
        3  {
    4,005      for (int i = 0; i < n; i++) {
    9,000          int min = find_min(arr, i, n-1);
7,014,178  => sorts.c:find_min (1000x)
   10,000          swap(&arr[i], &arr[min]);
   15,000  => sorts.c:swap (1000x)
        .    }
        2  }
        .
```

Basic Usage of Callgrind

Adding in cache simulation

- Invoke valgrind by --simulate-cache=yes option
  - valgrind --tool=callgrind --simulate-cache=yes name.out program-arguments
- The cache simulator models a machine with a split L1 cache (separate instruction I1 and data D1), backed by a unified second-level cache (L2).
- Similar to the previous example, callgrind_annotate should be used to interpret the output.

Output example

```
==16409== Events    : Ir Dr Dw I1mr D1mr D1mw I2mr D2mr D2mw
==16409== Collected : 7163066 4062243 537262 591 610 182 16 103 94
==16409==
==16409== I   refs:      7,163,066
==16409== I1  misses:          591
==16409== L2i misses:           16
==16409== I1  miss rate:       0.0%
==16409== L2i miss rate:       0.0%
==16409==
==16409== D   refs:      4,599,505  (4,062,243 rd + 537,262 wr)
==16409== D1  misses:          792  (      610 rd +     182 wr)
==16409== L2d misses:          197  (      103 rd +      94 wr)
==16409== D1  miss rate:       0.0% (      0.0%   +     0.0%  )
==16409== L2d miss rate:       0.0% (      0.0%   +     0.0%  )
==16409==
==16409== L2 refs:           1,383  (    1,201 rd +     182 wr)
==16409== L2 misses:           213  (      119 rd +      94 wr)
==16409== L2 miss rate:        0.0% (      0.0%   +     0.0%  )
```

It sounds like we have a cache friendly code.

Ir: I cache reads (instructions executed)

I1mr: I1 cache read misses (instruction wasn't in I1 cache but was in L2)

I2mr: L2 cache instruction read misses (instruction wasn't in I1 or L2 cache, had to be fetched

Dr: D cache reads (memory reads)

D1mr: D1 cache read misses (data location not in D1 cache, but in L2)

D2mr: L2 cache data read misses (location not in D1 or L2)

Dw: D cache writes (memory writes)

D1mw: D1 cache write misses (location not in D1 cache, but in L2)

D2mw: L2 cache data write misses (location not in D1 or L2)

KOÇ UNIVERSITY

Output example

```
--------------------------------------------------------------------------------
-- Auto-annotated source: sorts.c
--------------------------------------------------------------------------------
       Ir        Dr      Dw I1mr D1mr D1mw I2mr D2mr D2mw
        .         .       .    .    .    .    .    .    . void swap(int *a, int *b)
    3,000         0   1,000    1    0    0    1    .    . {
    3,000     2,000   1,000    .    .    .    .    .      int tmp = *a;
    4,000     3,000   1,000    .    .    .    .    .      *a = *b;
    3,000     2,000   1,000    .    .    .    .    .      *b = tmp;
    2,000     2,000       .    .    .    .    .    .    . }
        .         .       .    .    .    .    .    .    .
        .         .       .    .    .    .    .    .    . int find_min(int arr[], int start, int st
op)
    3,000         0   1,000    1    0    0    1    .    . {
    2,000     1,000   1,000    0    0    1    0    0    1      int min = start;
    2,005,000 1,002,000 500,500    .    .    .    .    .      for(int i = start+1; i <= st
op; i++)
4,995,000 2,997,000       0    0   32    0    0   19    .        if (arr[i] < arr[m
in])
    6,144     3,072   3,072    .    .    .    .    .    .           min = i;
    1,000     1,000       .    .    .    .    .    .    .    return min;
    2,000     2,000       .    .    .    .    .    .    . }
        .         .       .    .    .    .    .    .    . void selection_sort(int arr[], int n)
        3         0       1    1    0    0    1    .    . {
    4,005     2,002   1,001    .    .    .    .    .      for (int i = 0; i < n; i++) {
    9,000     3,000   5,000    .    .    .    .    .        int min = find_min(arr, i, n
-1);
    7,014,144 4,006,072 505,572    1   32    1    1   19    1  => sorts.c:find_min
(1000x)
   10,000     4,000   3,000    .    .    .    .    .        swap(&arr[i], &arr[min]);
   15,000     9,000   4,000    1    0    0    1    .    . => sorts.c:swap (1000x)
        .         .       .    .    .    .    .    .    }
        2         2       .    .    .    .    .    .    . }
```

Ir: I cache reads (instructions executed)

I1mr: I1 cache read misses (instruction wasn't in I1 cache but was in L2)

I2mr: L2 cache instruction read misses (instruction wasn't in I1 or L2 cache, had to be fetched

Dr: D cache reads (memory reads)

D1mr: D1 cache read misses (data location not in D1 cache, but in L2)

D2mr: L2 cache data read misses (location not in D1 or L2)

Dw: D cache writes (memory writes)

D1mw: D1 cache write misses (location not in D1 cache, but in L2)

D2mw: L2 cache data write misses (location not in D1 or L2)

Additional Points

- L2 misses are much more expensive than L1 misses, so pay attention to passages with high D2mr or D2mw counts.
- Even a small number of misses can be quite important, as a L1 miss will typically cost around 5-10 cycles, an L2 miss can cost as much as 100-200 cycles
- Callgrind cannot detect the bottleneck of your program if it is related to file I/O
- Try to examine different path of your program

References

1. Some of the slides are borrowed from materials in Stanford CS107, CMU15-213 and CS201, Portland State University
2. https://stackoverflow.com/questions/16699247/what-is-a-cache-friendly-code
3. https://www.valgrind.org/docs/manual/manual.html