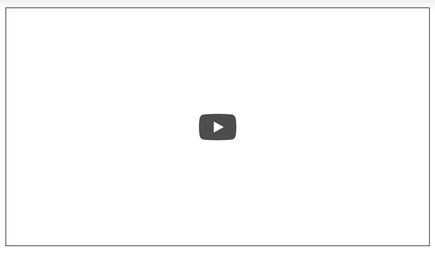
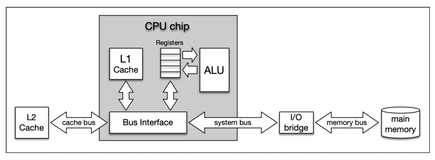


https://youtu.be/FNaaBipEqBw

Cache Levels

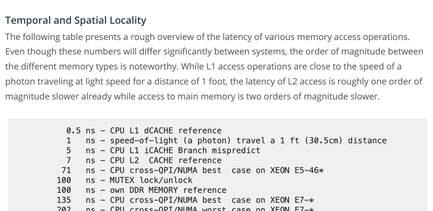
Cache memory is much faster but also significantly smaller than standard RAM. It holds the data that will (or might) be used by the CPU more often. In the memory hierarchy we have seen in the last section, the cache plays an intermediary role between fast CPU and slow RAM and hard disk. The figure below gives a rough overview of a typical system architecture:



System architecture diagram showing caches, ALU (arithmetic logic unit), main memory, and the buses connected each component.

The central CPU chip is connected to the outside world by a number of buses. There is a cache bus, which leads to a block denoted as L2 cache, and there is a system bus as well as a memory bus that leads to the computer main memory. The latter holds the comparatively large RAM while the L2 cache as well as the L1 cache are very small with the latter also being a part of the CPU itself.

The concept of L1 and L2 (and even L3) cache is further illustrated by the following figure, which shows a multi-core CPU and its interplay with L1, L2 and L3 caches:



L1, L2, and L3 cache

1. **Level 1 cache** is the fastest and smallest memory type in the cache hierarchy. In most systems, the L1 cache is not very large. Mostly it is in the range of 16 to 64 kBytes, where the memory areas for instructions and data are separated from each other (L1i and L1d, where "i" stands for "instruction" and "d" stands for "data". Also see "**Harvard architecture**" for further reference). The importance of the L1 cache grows with increasing speed of the CPU. In the L1 cache, the most frequently required instructions and data are buffered so that as few accesses as possible to the slow main memory are required. This cache avoids delays in data transmission and helps to make optimum use of the CPU's capacity.
2. **Level 2 cache** is located close to the CPU and has a direct connection to it. The information exchange between L2 cache and CPU is managed by the L2 controller on the computer main board. The size of the L2 cache is usually at or below 2 megabytes. On modern multi-core processors, the L2 cache is often located within the CPU itself. The choice between a processor with more clock speed or a larger L2 cache can be answered as follows: With a higher clock speed, individual programs run faster, especially those with high computing requirements. As soon as several programs run simultaneously, a larger cache is advantageous. Usually normal desktop computers with a processor that has a large cache are better served than with a processor that has a high clock rate.
3. **Level 3 cache** is shared among all cores of a multicore processor. With the L3 cache, the **cache coherence** protocol of multicore processors can work much faster. This protocol compares the caches of all cores to maintain data consistency so that all processors have access to the same data at the same time. The L3 cache therefore has less the function of a cache, but is intended to simplify and accelerate the cache coherence protocol and the data exchange between the cores.

On Mac, information about the system cache can be obtained by executing the command `sysctl -a hw` in a terminal. On Debian Linux linux, this information can be found with `lscpu | grep cache`. On my iMac Pro (2017), this command yielded (among others) the following output:

```
hw.memsize: 34359738368
hw.l1icachesize: 32768
hw.l1dcachesize: 32768
hw.l2cachesize: 1048576
hw.l3cachesize: 14417920
```

- *hw.l1icachesize* is the size of the L1 instruction cache, wich is at 32kB. This cache is strictly reserved for storing CPU instructions only.
- *hw.l1dcachesize* is also 32 KB and is dedicated for data as opposed to instructions.
- *hw.l2cachesize* and *hw.l3cachesize* show the size of the L2 and L3 cache, which are at 1MB and 14MB respectively.

It should be noted that the size of all caches combined is very small when compared to the size of the main memory (the RAM), which is at 32GB on my system.

Ideally, data needed by the CPU should be read from the various caches for more than 90% of all memory access operations. This way, the high latency of RAM and hard disk can be efficiently compensated.

Temporal and Spatial Locality

The following table presents a rough overview of the latency of various memory access operations. Even though these numbers will differ significantly between systems, the order of magnitude between the different memory types is noteworthy. While L1 access operations are close to the speed of a photon traveling at light speed for a distance of 1 foot, the latency of L2 access is roughly one order of magnitude slower already while access to main memory is two orders of magnitude slower.

```
          0.5 ns - CPU L1 dCACHE reference
            1   ns - speed-of-light (a photon) travel a 1 ft (30.5cm) distance
            5   ns - CPU L1 iCACHE Branch mispredict
            7   ns - CPU L2  CACHE reference
           71   ns - CPU cross-QPI/NUMA best  case on XEON E7-*
          100   ns - MUTEX lock/unlock
          100   ns - own DDR MEMORY reference
          135   ns - CPU cross-QPI/NUMA best  case on XEON E7-*
          202   ns - CPU cross-QPI/NUMA worst case on XEON E7-*
          325   ns - CPU cross-QPI/NUMA worst case on XEON E5-46*
       10,000   ns - Compress 1K bytes with Zippy PROCESS
       20,000   ns - Send 2K bytes over 1 Gbps NETWORK
      250,000   ns - Read 1 MB sequentially from MEMORY
      500,000   ns - Round trip within a same DataCenter
   10,000,000   ns - DISK seek
   10,000,000   ns - Read 1 MB sequentially from NETWORK
   30,000,000   ns - Read 1 MB sequentially from DISK
  150,000,000   ns - Send a NETWORK packet CA -> Netherlands
|   |    |    |    |
|   |    |    | ns|
|   |    | us|
| ms|
```

Originally from Peter Norvig: http://norvig.com/21-days.html#answers

In algorithm design, programmers can exploit two principles to increase runtime performance:

1. **Temporal locality** means that address ranges that are accessed are likely to be used again in the near future. In the course of time, the same memory address is accessed relatively frequently (e.g. in a loop). This property can be used at all levels of the memory hierarchy to keep memory areas accessible as quickly as possible.
2. **Spatial locality** means that after an access to an address range, the next access to an address in the immediate vicinity is highly probable (e.g. in arrays). In the course of time, memory addresses that are very close to each other are accessed again multiple times. This can be exploited by moving the adjacent address areas upwards into the next hierarchy level during a memory access.

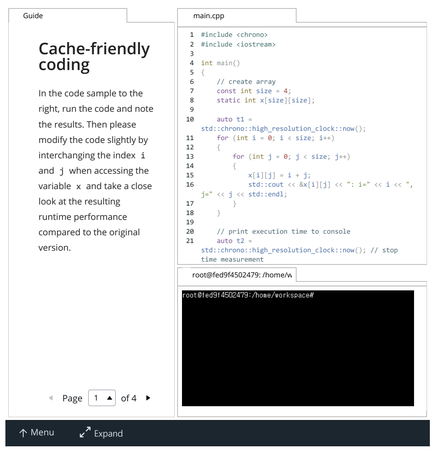
Let us consider the following code example:

```
#include <chrono>
#include <iostream>

int main()
{
    // create array
    const int size = 4;
    static int x[size][size];

    auto t1 = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            x[j][i] = i + j;
            std::cout << &x[j][i] << ": i=" << i << ", j=" << j << std::endl;
        }
    }

    // print execution time to console
    auto t2 = std::chrono::high_resolution_clock::now(); // stop time measurement
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(t2 - t1).count();
    std::cout << "Execution time: " << duration << " microseconds" << std::endl;

    return 0;
}
```

https://youtu.be/Gfrfp1GtIUU

Exercise : Cache-friendly coding

> Note: Click "Expand" at the bottom left of the workspace below for better readability and overall workspace experience.



Outro

https://youtu.be/izXHpEad7hY