```
struct D4 : S4
{
    char d;     // size 1; alignment 1
};              // size 3; alignment 1
```

Finally, virtual functions and virtual base classes invariably introduce an implicit virtual-table-pointer member having a size and alignment corresponding to that of a memory address (e.g., 4 or 8) on the target platform:

```
struct S5
{
    virtual ~S5();
};              // size 8; alignment 8

struct D5 : S5
{
    char d;     // size  1; alignment 1
};              // size 16; alignment 8
```

Cache lines; L1, L2, and L3 cache; pages; and virtual memory

Modern computers are highly complex systems, and a detailed understanding of their intricacies is unnecessary to achieve most of the performance benefits. Still, certain general themes and rough thresholds aid in understanding how to squeeze just a bit more out of the underlying hardware. In this section, we sketch fundamental concepts that are common to all modern computer hardware; although the precise details will vary, the general ideas remain essentially the same.

In its most basic form, a computer consists of central processing unit (CPU) having internal registers that access main memory (MM). Registers in the CPU (on the order of hundreds of bytes) are among the fastest forms of memory, while MM, typically many gigabytes, is orders of magnitude slower. An almost universally observed phenomenon is that of **locality of reference**, which suggests that data that resides in close proximity in the virtual address space is more likely to be accessed together in rapid succession than more distant data.

To exploit the phenomenon of locality of reference, computers introduce the notion of a cache that, while much faster than MM, is also much smaller. Programs that attempt to amplify locality of reference will, in turn, often be rewarded with faster run times. The organization of a cache and, in fact, the number of levels of cache, e.g., L1, L2, L3,..., will vary, but the basic design parameters are, again, more or less the same. A given level of cache will have a certain total size in bytes, invariably an integral power of two. The cache will be segmented into what are called cache lines whose size — a smaller power of two — divides that of the cache itself. When the CPU accesses MM, it first looks to see if that memory is in the cache; if it is, the value is returned quickly, known as a **cache hit**. Otherwise, the cache lines containing that data are fetched from the next higher level of cache or from MM

page reflows

and placed into the cache (known as a **cache miss**), possibly ejecting other less recently used ones.[11]

Data residing in distinct cache lines is physically independent and can be written concurrently by multiple threads, possibly running on separate cores or even processors. Logically unrelated data residing in the same cache line, however, is nonetheless physically coupled; two threads that write to such logically unrelated data will find themselves synchronized by the hardware. Such unexpected and typically undesirable sharing of a cache line by unrelated data acted upon by two concurrent threads is known as false sharing. One way of avoiding false sharing is to align such data on a cache-line boundary, thus rendering impossible the accidental colocation of such data on the same cache line. Another, more broad-based design approach that avoids lowering cache utilization is to ensure that data acted upon by a given thread is kept physically separate, e.g., through the use of local, arena memory allocators.[12]

Finally, even data that is not currently in cache but resides nearby in ~~MM~~ can benefit from locality. The virtual address space~~, synonymous with the size of a **void***~~ (typically ~~64 bits~~ on modern general-purpose hardware), has historically well exceeded the physical memory available to the CPU. The operating system must therefore maintain a mapping in ~~MM~~ from what is resident in physical memory and what resides in secondary storage, e.g., on ~~disc~~. In addition, essentially all modern hardware provides a translation-lookaside buffer (TLB)[13] that caches the addresses of the most recently accessed physical pages, providing yet another advantage to having the **working set**, i.e., the current set of frequently accessed

page reflows

[11] Conceptually, the cache is often thought of as being able to hold any arbitrary subset of the most recently accessed cache lines. This kind of cache is known as **fully associative**. Although it provides the best hit rate, a fully associative cache requires the most power along with significant additional chip area to perform the fully parallel lookup. **Direct-mapped** cache associativity is at the other extreme. In direct mapped, each memory location has exactly one location available to it in the cache. If another memory location mapping to that location is needed, the current cache line must be flushed from the cache. This approach has the lowest hit rate, but lookup times, chip area, and power consumption are all optimally minimized. Between these two extremes is a continuum that is referred to as **set associative**. A set ~~associate~~ cache has more than one — typically 2, 4, or 8; see **solihin15**, section 5.2.1, "Placement Policy," pp. 136–141, and **hruska20** — location in which each memory location in main memory can reside. Note that, even with a relatively small N, as N increases, an N-way set associative cache quickly approaches the hit rate of a fully associative cache at greatly reduced collateral cost; for most software-design purposes, any loss in hit rate due to set associativity of a cache can be safely ignored.

[12] **lakos17b**, **lakos19**, **lakos22**

[13] A TLB is a kind of address-translation cache that is typically part of a chip's memory management unit. A TLB holds a recently accessed subset of the complete mapping, itself maintained in ~~MM~~, from virtual memory address to physical ones. A TLB is used to reduce access time when the requisite pages are already resident in memory; its size, e.g., 4K, is capped at the number of bytes of physical memory, e.g., 32Gb, divided by the number of bytes in each physical page, e.g., 8Kb, but could be smaller. Because it resides on chip, is typically an order of magnitude faster (SRAM versus DRAM), and requires only a single lookup (as opposed to two or more when going out to MM), there is an enormous premium on minimizing TLB misses.

pages, remain small and densely packed with relevant data.[14] What's more, dense working sets, in addition to facilitating hits for repeat access, increase the likelihood that data that is coresident on a page or cache line will be needed soon, i.e., in effect acting as a prefetch. Table 1 provides a summary of typical physical parameters found in modern computers today.

Table 1: Various sizes and access speeds of typical memory for modern computers

Memory Type	Typical Memory Size (Bytes)	Typical Access Times
CPU Registers	512 ... 2048	~250ps
Cache Line	64 ... 256	NA
L1 Cache	16Kb ... 64Kb	~1ns
L2 Cache	1Mb ... 2Mb	~10ns
L3 Cache	8Mb ... 32Mb	~80ns ... 120ns
L4 Cache	32Mb ... 128Mb	~100ns ... 200ns
Set Associativity	2 ... 64	NA
TL	4 words ... 65536 words	~10ns ... 50ns
Physical Memory Page	512 ... 8192	~100ns ... 500ns
Virtual Memory	2^{32} bytes ... 2^{64} bytes	~10μs ... 50μs
Solid-State Disc (SSD)	256Gb ... 16Tb	~25μs ... 100μs
Mechanical Disc	Huge	~5ms ... 10ms
Clock Speed	NA	~4GHz

[14] Note that memory for handle-body types (e.g., `std::vector` or `std::deque`) and especially node-based containers (e.g., `std::map` and `std::unordered_map`), originally allocated within a single page, can — through deallocation and reallocation or even move operations — become scattered across multiple, perhaps many, pages, thus causing what was originally a relatively small working set to no longer fit within physical memory. This phenomenon, known as **diffusion** (which is a distinct concept from **fragmentation**), is what typically leads to a substantial runtime performance degradation due to cache line **thrashing** in large, long-running programs. Such diffusion can be mitigated by judicious use of local arena memory allocators and deliberate avoidance of **move operations** across disparate localities of frequent memory usage.