1.3 Memory hierarchies

Data can be stored in a computer system in many different ways. As described above, CPUs have a set of registers, which can be accessed without delay. In addition there are one or more small but very fast *caches* holding copies of recently used data items. *Main memory* is much slower, but also much larger than cache. Finally, data can be stored on disk and copied to main memory as needed. This a is a complex hierarchy, and it is vital to understand how data transfer works between the different levels in order to identify performance bottlenecks. In the following we will concentrate on all levels from CPU to main memory (see Figure 1.3).

1.3.1 Cache

Caches are low-capacity, high-speed memories that are commonly integrated on the CPU die. The need for caches can be easily understood by realizing that data transfer rates to main memory are painfully slow compared to the CPU's arithmetic performance. While peak performance soars at several GFlops/sec per core, *memory bandwidth*, i.e., the rate at which data can be transferred from memory to the CPU, is still stuck at a couple of GBytes/sec, which is entirely insufficient to feed all arithmetic units and keep them busy continuously (see Chapter 3 for a more thorough analysis). To make matters worse, in order to transfer a single data item (usually one or two DP words) from memory, an initial waiting time called *latency* passes until data can actually flow. Thus, latency is often defined as the time it takes to transfer a zero-byte message. Memory latency is usually of the order of several hundred CPU cycles and is composed of different contributions from memory chips, the chipset and the processor. Although Moore's Law still guarantees a constant rate of improvement in chip complexity and (hopefully) performance, advances in memory performance show up at a much slower rate. The term *DRAM gap* has been coined for the increasing "distance" between CPU and memory in terms of latency and bandwidth [R34, R37].

Caches can alleviate the effects of the DRAM gap in many cases. Usually there are at least two *levels* of cache (see Figure 1.3), called *L1* and *L2*, respectively. L1 is normally split into two parts, one for instructions ("I-cache," "L1I") and one for data ("L1D"). Outer cache levels are normally *unified*, storing data as well as instructions. In general, the "closer" a cache is to the CPU's registers, i.e., the higher its bandwidth and the lower its latency, the smaller it must be to keep administration overhead low. Whenever the CPU issues a read request ("load") for transferring a data item to a register, first-level cache logic checks whether this item already resides in cache. If it does, this is called a *cache hit* and the request can be satisfied immediately, with low latency. In case of a *cache miss*, however, data must be fetched from outer cache levels or, in the worst case, from main memory. If all cache entries are occupied, a hardware-implemented algorithm *evicts* old items from cache and replaces them with new data. The sequence of events for a cache miss on a write is more involved and

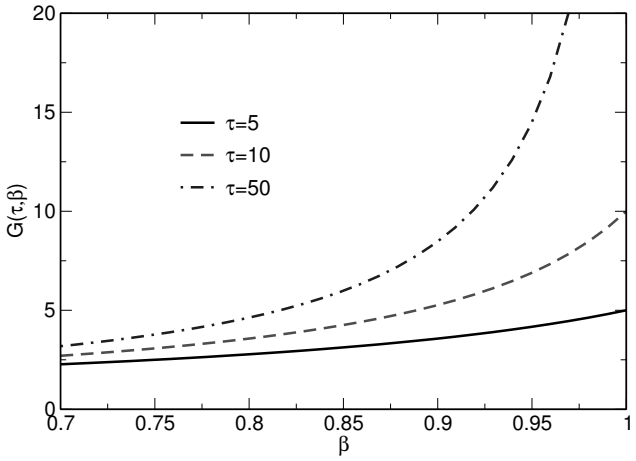

Figure 1.9: The performance gain from accessing data from cache versus the cache reuse ratio, with the speed advantage of cache versus main memory being parametrized by τ.

will be described later. Instruction caches are usually of minor importance since scientific codes tend to be largely loop-based; I-cache misses are rare events compared to D-cache misses.

Caches can only have a positive effect on performance if the data access pattern of an application shows some *locality of reference*. More specifically, data items that have been loaded into a cache are to be used again "soon enough" to not have been evicted in the meantime. This is also called *temporal locality*. Using a simple model, we will now estimate the performance gain that can be expected from a cache that is a factor of τ faster than memory (this refers to bandwidth as well as latency; a more refined model is possible but does not lead to additional insight). Let β be the *cache reuse ratio*, i.e., the fraction of loads or stores that can be satisfied from cache because there was a recent load or store to the same address. Access time to main memory (again this includes latency and bandwidth) is denoted by T_m. In cache, access time is reduced to $T_\mathrm{c} = T_\mathrm{m}/\tau$. For some finite β, the average access time will thus be $T_\mathrm{av} = \beta T_\mathrm{c} + (1-\beta)T_\mathrm{m}$, and we calculate an access performance gain of

$$G(\tau,\beta) = \frac{T_\mathrm{m}}{T_\mathrm{av}} = \frac{\tau T_\mathrm{c}}{\beta T_\mathrm{c} + (1-\beta)\tau T_\mathrm{c}} = \frac{\tau}{\beta + \tau(1-\beta)} \,. \tag{1.4}$$

As Figure 1.9 shows, a cache can only lead to a significant performance advantage if the reuse ratio is relatively close to one.

Unfortunately, supporting temporal locality is not sufficient. Many applications show *streaming* patterns where large amounts of data are loaded into the CPU, modified, and written back without the potential of reuse "in time." For a cache that only supports temporal locality, the reuse ratio β (see above) is zero for streaming. Each new load is expensive as an item has to be evicted from cache and replaced by the new one, incurring huge latency. In order to reduce the latency penalty for streaming, caches feature a peculiar organization into *cache lines*. All data transfers between caches and main memory happen on the cache line level (there may be exceptions from that rule; see the comments on nontemporal stores on page 18 for details). The

advantage of cache lines is that the latency penalty of a cache miss occurs only on the first miss on an item belonging to a line. The line is fetched from memory as a whole; neighboring items can then be loaded from cache with much lower latency, increasing the *cache hit ratio* γ, not to be confused with the reuse ratio β. So if the application shows some *spatial locality*, i.e., if the probability of successive accesses to neighboring items is high, the latency problem can be significantly reduced. The downside of cache lines is that erratic data access patterns are not supported. On the contrary, not only does each load incur a miss and subsequent latency penalty, it also leads to the transfer of a whole cache line, polluting the memory bus with data that will probably never be used. The effective bandwidth available to the application will thus be very low. On the whole, however, the advantages of using cache lines prevail, and very few processor manufacturers have provided means of bypassing the mechanism.

Assuming a streaming application working on DP floating point data on a CPU with a cache line length of $L_c = 16$ words, spatial locality fixes the hit ratio at $\gamma = (16-1)/16 = 0.94$, a seemingly large value. Still it is clear that performance is governed by main memory bandwidth and latency — the code is *memory-bound*. In order for an application to be truly *cache-bound*, i.e., decouple from main memory so that performance is not governed by memory bandwidth or latency any more, γ must be large enough so the time it takes to process in-cache data becomes larger than the time for reloading it. If and when this happens depends of course on the details of the operations performed.

By now we can qualitatively interpret the performance data for cache-based architectures on the vector triad in Figure 1.4. At very small loop lengths, the processor pipeline is too long to be efficient. With growing `N` this effect becomes negligible, and as long as all four arrays fit into the innermost cache, performance saturates at a high value that is set by the L1 cache bandwidth and the ability of the CPU to issue load and store instructions. Increasing `N` a little more gives rise to a sharp drop in performance because the innermost cache is not large enough to hold all data. Second-level cache has usually larger latency but similar bandwidth to L1 so that the penalty is larger than expected. However, streaming data from L2 has the disadvantage that L1 now has to provide data for registers as well as continuously reload and evict cache lines from/to L2, which puts a strain on the L1 cache's bandwidth limits. Since the ability of caches to deliver data to higher and lower hierarchy levels concurrently is highly architecture-dependent, performance is usually hard to predict on all but the innermost cache level and main memory. For each cache level another performance drop is observed with rising `N`, until finally even the large outer cache is too small and all data has to be streamed from main memory. The size of the different caches is directly related to the locations of the bandwidth breakdowns. Section 3.1 will describe how to predict performance for simple loops from basic parameters like cache or memory bandwidths and the data demands of the application.

Storing data is a little more involved than reading. In presence of caches, if data to be written out already resides in cache, a *write hit* occurs. There are several possibilities for handling this case, but usually outermost caches work with a *write-back* strategy: The cache line is modified in cache and written to memory as a whole when

evicted. On a *write miss*, however, cache-memory consistency dictates that the cache line in question must first be transferred from memory to cache before an entry can be modified. This is called *write allocate*, and leads to the situation that a data write stream from CPU to memory uses the bus twice: once for all the cache line allocations and once for evicting modified lines (the data transfer requirement for the triad benchmark code is increased by 25% due to write allocates). Consequently, streaming applications do not usually profit from write-back caches and there is often a wish for avoiding write-allocate transactions. Some architectures provide this option, and there are generally two different strategies:

- *Nontemporal stores*. These are special store instructions that bypass all cache levels and write directly to memory. Cache does not get "polluted" by store streams that do not exhibit temporal locality anyway. In order to prevent excessive latencies, there is usually a small *write combine buffer*, which bundles a number of successive nontemporal stores [V104].

- *Cache line zero*. Special instructions "zero out" a cache line and mark it as modified without a prior read. The data is written to memory when evicted. In comparison to nontemporal stores, this technique uses up cache space for the store stream. On the other hand it does not slow down store operations in cache-bound situations. Cache line zero must be used with extreme care: All elements of a cache line are evicted to memory, even if only a part of them were actually modified.

Both can be applied by the compiler and hinted at by the programmer by means of directives. In very simple cases compilers are able to apply those instructions automatically in their optimization stages, but one must take care to not slow down a cache-bound code by using nontemporal stores, rendering it effectively memory-bound.

Note that the need for write allocates arises because caches and memory generally communicate in units of cache lines; it is a common misconception that write allocates are only required to maintain consistency between caches of multiple processor cores.

1.3.2 Cache mapping

So far we have implicitly assumed that there is no restriction on which cache line can be associated with which memory locations. A cache design that follows this rule is called *fully associative*. Unfortunately it is quite hard to build large, fast, and fully associative caches because of large bookkeeping overhead: For each cache line the cache logic must store its location in the CPU's address space, and each memory access must be checked against the list of all those addresses. Furthermore, the decision which cache line to replace next if the cache is full is made by some algorithm implemented in hardware. Often there is a *least recently used* (LRU) strategy that makes sure only the "oldest" items are evicted, but alternatives like NRU (*not recently used*) or random replacement are possible.