7.4.3 Locality in Programs

Most programs exhibit a high degree of *locality*; that is, they spend most of their time executing a relatively small fraction of the code and touching only a small fraction of the data. We say that a program has *temporal locality* if the memory locations it accesses are likely to be accessed again within a short period of time. We say that a program has *spatial locality* if memory locations close to the location accessed are likely also to be accessed within a short period of time.

The conventional wisdom is that programs spend 90% of their time executing 10% of the code. Here is why:

- Programs often contain many instructions that are never executed. Programs built with components and libraries use only a small fraction of the provided functionality. Also as requirements change and programs evolve, legacy systems often contain many instructions that are no longer used.

Static and Dynamic RAM

Most random-access memory is *dynamic*, which means that it is built of very simple electronic circuits that lose their charge (and thus "forget" the bit they were storing) in a short time. These circuits need to be refreshed — that is, their bits read and rewritten — periodically. On the other hand, *static* RAM is designed with a more complex circuit for each bit, and consequently the bit stored can stay indefinitely, until it is changed. Evidently, a chip can store more bits if it uses dynamic-RAM circuits than if it uses static-RAM circuits, so we tend to see large main memories of the dynamic variety, while smaller memories, like caches, are made from static circuits.

- Only a small fraction of the code that could be invoked is actually executed in a typical run of the program. For example, instructions to handle illegal inputs and exceptional cases, though critical to the correctness of the program, are seldom invoked on any particular run.
- The typical program spends most of its time executing innermost loops and tight recursive cycles in a program.

Locality allows us to take advantage of the memory hierarchy of a modern computer, as shown in Fig. 7.16. By placing the most common instructions and data in the fast-but-small storage, while leaving the rest in the slow-but-large storage, we can lower the average memory-access time of a program significantly.

It has been found that many programs exhibit both temporal and spatial locality in how they access both instructions and data. Data-access patterns, however, generally show a greater variance than instruction-access patterns. Policies such as keeping the most recently used data in the fastest hierarchy work well for common programs but may not work well for some data-intensive programs — ones that cycle through very large arrays, for example.

We often cannot tell, just from looking at the code, which sections of the code will be heavily used, especially for a particular input. Even if we know which instructions are executed heavily, the fastest cache often is not large enough to hold all of them at the same time. We must therefore adjust the contents of the fastest storage dynamically and use it to hold instructions that are likely to be used heavily in the near future.

Optimization Using the Memory Hierarchy

The policy of keeping the most recently used instructions in the cache tends to work well; in other words, the past is generally a good predictor of future memory usage. When a new instruction is executed, there is a high probability that the next instruction also will be executed. This phenomenon is an

Cache Architectures

How do we know if a cache line is in a cache? It would be too expensive to check every single line in the cache, so it is common practice to restrict the placement of a cache line within the cache. This restriction is known as *set associativity*. A cache is *k-way set associative* if a cache line can reside only in k locations. The simplest cache is a 1-way associative cache, also known as a *direct-mapped* cache. In a direct-mapped cache, data with memory address n can be placed only in cache address n mod s, where s is the size of the cache. Similarly, a k-way set associative cache is divided into k sets, where a datum with address n can be mapped only to the location n mod (s/k) in each set. Most instruction and data caches have associativity between 1 and 8. When a cache line is brought into the cache, and all the possible locations that can hold the line are occupied, it is typical to evict the line that has been the least recently used.

example of spatial locality. One effective technique to improve the spatial locality of instructions is to have the compiler place basic blocks (sequences of instructions that are always executed sequentially) that are likely to follow each other contiguously — on the same page, or even the same cache line, if possible. Instructions belonging to the same loop or same function also have a high probability of being executed together.[4]

We can also improve the temporal and spatial locality of data accesses in a program by changing the data layout or the order of the computation. For example, programs that visit large amounts of data repeatedly, each time performing a small amount of computation, do not perform well. It is better if we can bring some data from a slow level of the memory hierarchy to a faster level (e.g., disk to main memory) once, and perform all the necessary computations on this data while it resides at the faster level. This concept can be applied recursively to reuse data in physical memory, in the caches and in the registers.