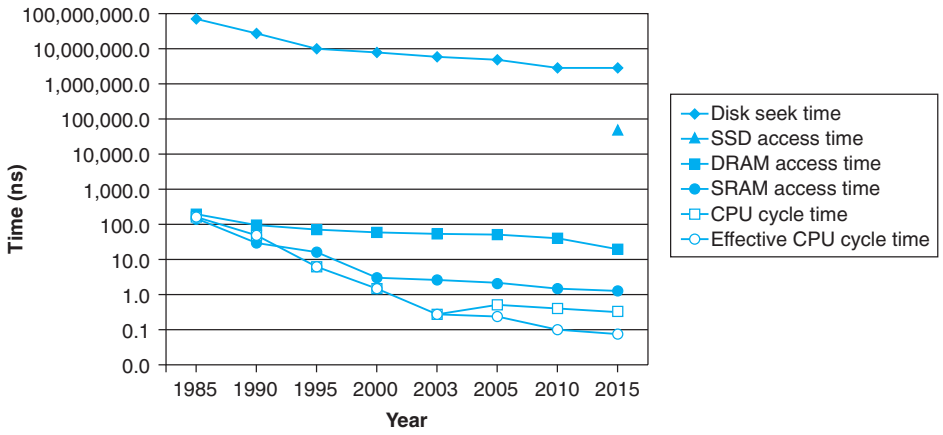

Figure 6.16 **The gap between disk, DRAM, and CPU speeds.**

throughput, with multiple processor cores issuing requests to the DRAM and disk in parallel.

The various trends are shown quite clearly in Figure 6.16, which plots the access and cycle times from Figure 6.15 on a semi-log scale.

As we will see in Section 6.4, modern computers make heavy use of SRAM-based caches to try to bridge the processor–memory gap. This approach works because of a fundamental property of application programs known as *locality*, which we discuss next.

Practice Problem 6.6 **(solution page 698)**

Using the data from the years 2005 to 2015 in Figure 6.15(c), estimate the year when you will be able to buy a petabyte (10^{15} bytes) of rotating disk storage for $200. Assume actual dollars (no inflation).

6.2 Locality

Well-written computer programs tend to exhibit good *locality*. That is, they tend to reference data items that are near other recently referenced data items or that were recently referenced themselves. This tendency, known as the *principle of locality*, is an enduring concept that has enormous impact on the design and performance of hardware and software systems.

Locality is typically described as having two distinct forms: *temporal locality* and *spatial locality*. In a program with good temporal locality, a memory location that is referenced once is likely to be referenced again multiple times in the near future. In a program with good spatial locality, if a memory location is referenced

Aside When cycle time stood still: The advent of multi-core processors

The history of computers is marked by some singular events that caused profound changes in the industry and the world. Interestingly, these inflection points tend to occur about once per decade: the development of Fortran in the 1950s, the introduction of the IBM 360 in the early 1960s, the dawn of the Internet (then called ARPANET) in the early 1970s, the introduction of the IBM PC in the early 1980s, and the creation of the World Wide Web in the early 1990s.

The most recent such event occurred early in the 21st century, when computer manufacturers ran headlong into the so-called power wall, discovering that they could no longer increase CPU clock frequencies as quickly because the chips would then consume too much power. The solution was to improve performance by replacing a single large processor with multiple smaller processor *cores*, each a complete processor capable of executing programs independently and in parallel with the other cores. This *multi-core* approach works in part because the power consumed by a processor is proportional to $P = fCV^2$, where f is the clock frequency, C is the capacitance, and V is the voltage. The capacitance C is roughly proportional to the area, so the power drawn by multiple cores can be held constant as long as the total area of the cores is constant. As long as feature sizes continue to shrink at the exponential Moore's Law rate, the number of cores in each processor, and thus its effective performance, will continue to increase.

From this point forward, computers will get faster not because the clock frequency increases but because the number of cores in each processor increases, and because architectural innovations increase the efficiency of programs running on those cores. We can see this trend clearly in Figure 6.16. CPU cycle time reached its lowest point in 2003 and then actually started to rise before leveling off and starting to decline again at a slower rate than before. However, because of the advent of multi-core processors (dual-core in 2004 and quad-core in 2007), the effective cycle time continues to decrease at close to its previous rate.

once, then the program is likely to reference a nearby memory location in the near future.

Programmers should understand the principle of locality because, in general, *programs with good locality run faster than programs with poor locality*. All levels of modern computer systems, from the hardware, to the operating system, to application programs, are designed to exploit locality. At the hardware level, the principle of locality allows computer designers to speed up main memory accesses by introducing small fast memories known as *cache memories* that hold blocks of the most recently referenced instructions and data items. At the operating system level, the principle of locality allows the system to use the main memory as a cache of the most recently referenced chunks of the virtual address space. Similarly, the operating system uses main memory to cache the most recently used disk blocks in the disk file system. The principle of locality also plays a crucial role in the design of application programs. For example, Web browsers exploit temporal locality by caching recently referenced documents on a local disk. High-volume Web servers hold recently requested documents in front-end disk caches that satisfy requests for these documents without requiring any intervention from the server.

```
1  int sumvec(int v[N])
2  {
3      int i, sum = 0;
4
5      for (i = 0; i < N; i++)
6          sum += v[i];
7      return sum;
8  }
```

(a)

Address	0	4	8	12	16	20	24	28
Contents	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
Access order	1	2	3	4	5	6	7	8

(b)

Figure 6.17 **(a) A function with good locality. (b) Reference pattern for vector** `v` **($N = 8$).** Notice how the vector elements are accessed in the same order that they are stored in memory.

6.2.1 Locality of References to Program Data

Consider the simple function in Figure 6.17(a) that sums the elements of a vector. Does this function have good locality? To answer this question, we look at the reference pattern for each variable. In this example, the `sum` variable is referenced once in each loop iteration, and thus there is good temporal locality with respect to `sum`. On the other hand, since `sum` is a scalar, there is no spatial locality with respect to `sum`.

As we see in Figure 6.17(b), the elements of vector `v` are read sequentially, one after the other, in the order they are stored in memory (we assume for convenience that the array starts at address 0). Thus, with respect to variable `v`, the function has good spatial locality but poor temporal locality since each vector element is accessed exactly once. Since the function has either good spatial or temporal locality with respect to each variable in the loop body, we can conclude that the `sumvec` function enjoys good locality.

A function such as `sumvec` that visits each element of a vector sequentially is said to have a *stride-1 reference pattern* (with respect to the element size). We will sometimes refer to stride-1 reference patterns as *sequential reference patterns*. Visiting every kth element of a contiguous vector is called a *stride-k reference pattern*. Stride-1 reference patterns are a common and important source of spatial locality in programs. In general, as the stride increases, the spatial locality decreases.

Stride is also an important issue for programs that reference multidimensional arrays. For example, consider the `sumarrayrows` function in Figure 6.18(a) that sums the elements of a two-dimensional array.

The doubly nested loop reads the elements of the array in *row-major order*. That is, the inner loop reads the elements of the first row, then the second row, and so on. The `sumarrayrows` function enjoys good spatial locality because it references the array in the same row-major order that the array is stored (Figure 6.18(b)). The result is a nice stride-1 reference pattern with excellent spatial locality.

```
1  int sumarrayrows(int a[M][N])
2  {
3      int i, j, sum = 0;
4
5      for (i = 0; i < M; i++)
6          for (j = 0; j < N; j++)
7              sum += a[i][j];
8      return sum;
9  }
```

(a)

Address	0	4	8	12	16	20
Contents	a_{00}	a_{01}	a_{02}	a_{10}	a_{11}	a_{12}
Access order	1	2	3	4	5	6

(b)

Figure 6.18 **(a) Another function with good locality. (b) Reference pattern for array** a **($M = 2$, $N = 3$).** There is good spatial locality because the array is accessed in the same row-major order in which it is stored in memory.

```
1  int sumarraycols(int a[M][N])
2  {
3      int i, j, sum = 0;
4
5      for (j = 0; j < N; j++)
6          for (i = 0; i < M; i++)
7              sum += a[i][j];
8      return sum;
9  }
```

(a)

Address	0	4	8	12	16	20
Contents	a_{00}	a_{01}	a_{02}	a_{10}	a_{11}	a_{12}
Access order	1	3	5	2	4	6

(b)

Figure 6.19 **(a) A function with poor spatial locality. (b) Reference pattern for array** a **($M = 2$, $N = 3$).** The function has poor spatial locality because it scans memory with a stride-N reference pattern.

Seemingly trivial changes to a program can have a big impact on its locality. For example, the `sumarraycols` function in Figure 6.19(a) computes the same result as the `sumarrayrows` function in Figure 6.18(a). The only difference is that we have interchanged the i and j loops. What impact does interchanging the loops have on its locality?

The `sumarraycols` function suffers from poor spatial locality because it scans the array column-wise instead of row-wise. Since C arrays are laid out in memory row-wise, the result is a stride-N reference pattern, as shown in Figure 6.19(b).

6.2.2 Locality of Instruction Fetches

Since program instructions are stored in memory and must be fetched (read) by the CPU, we can also evaluate the locality of a program with respect to its instruction fetches. For example, in Figure 6.17 the instructions in the body of the

`for` loop are executed in sequential memory order, and thus the loop enjoys good spatial locality. Since the loop body is executed multiple times, it also enjoys good temporal locality.

An important property of code that distinguishes it from program data is that it is rarely modified at run time. While a program is executing, the CPU reads its instructions from memory. The CPU rarely overwrites or modifies these instructions.

6.2.3 Summary of Locality

In this section, we have introduced the fundamental idea of locality and have identified some simple rules for qualitatively evaluating the locality in a program:

- Programs that repeatedly reference the same variables enjoy good temporal locality.
- For programs with stride-k reference patterns, the smaller the stride, the better the spatial locality. Programs with stride-1 reference patterns have good spatial locality. Programs that hop around memory with large strides have poor spatial locality.
- Loops have good temporal and spatial locality with respect to instruction fetches. The smaller the loop body and the greater the number of loop iterations, the better the locality.

Later in this chapter, after we have learned about cache memories and how they work, we will show you how to quantify the idea of locality in terms of cache hits and misses. It will also become clear to you why programs with good locality typically run faster than programs with poor locality. Nonetheless, knowing how to glance at a source code and getting a high-level feel for the locality in the program is a useful and important skill for a programmer to master.

Practice Problem 6.7 **(solution page 698)**

Permute the loops in the following function so that it scans the three-dimensional array a with a stride-1 reference pattern.

```
1   int productarray3d(int a[N][N][N])
2   {
3       int i, j, k, product = 1;
4
5       for (i = N-1; i >= 0; i--) {
6           for (j = N-1; j >= 0; j--) {
7               for (k = N-1; k >= 0; k--) {
8                   product *= a[j][k][i];
9               }
10          }
11      }
12      return product;
13  }
```

(a) An array of `structs`

```
#define N 1000

typedef struct {
    int vel[3];
    int acc[3];
} point;

point p[N];
```

(b) The `clear1` function

```
void clear1(point *p, int n)
{
    int i, j;

    for (i = 0; i < n; i++) {
        for (j = 0; j < 3; j++)
            p[i].vel[j] = 0;
        for (j = 0; j < 3; j++)
            p[i].acc[j] = 0;
    }
}
```

(c) The `clear2` function

```
void clear2(point *p, int n)
{
    int i, j;

    for (i = 0; i < n; i++) {
        for (j = 0; j < 3; j++) {
            p[i].vel[j] = 0;
            p[i].acc[j] = 0;
        }
    }
}
```

(d) The `clear3` function

```
void clear3(point *p, int n)
{
    int i, j;

    for (j = 0; j < 3; j++) {
        for (i = 0; i < n; i++)
            p[i].vel[j] = 0;
        for (i = 0; i < n; i++)
            p[i].acc[j] = 0;
    }
}
```

Figure 6.20 **Code examples for Practice Problem 6.8.**

Practice Problem 6.8 **(solution page 699)**

The three functions in Figure 6.20 perform the same operation with varying degrees of spatial locality. Rank-order the functions with respect to the spatial locality enjoyed by each. Explain how you arrived at your ranking.

6.3 The Memory Hierarchy

Sections 6.1 and 6.2 described some fundamental and enduring properties of storage technology and computer software:

Storage technology. Different storage technologies have widely different access times. Faster technologies cost more per byte than slower ones and have less capacity. The gap between CPU and main memory speed is widening.

Computer software. Well-written programs tend to exhibit good locality.