6 Storage Systems

I/O certainly has been lagging in the last decade.

Seymour Cray
Public Lecture (1976)

Also, I/O needs a lot of work.

David Kuck
Keynote Address, 15th Annual Symposium on Computer Architecture (1988)

6.1 Introduction

Input/output has been the orphan of computer architecture. Historically neglected by CPU enthusiasts, the prejudice against I/O is institutionalized in the most widely used performance measure, CPU time (page 32). The quality of a computer's I/O system—whether it has the best or worst in the world—cannot be measured by CPU time, which by definition ignores I/O. The second-class citizenship of I/O is even apparent in the label *peripheral* applied to I/O devices.

This attitude is contradicted by common sense. A computer without I/O devices is like a car without wheels—you can't get very far without them. And while CPU time is interesting, response time—the time between when the user types a command and when results appear—is surely a better measure of performance. The customer who pays for a computer cares about response time, even if the CPU designer doesn't.

I/O's revenge is at hand. Suppose we have a difference between CPU time and response time of 10%, and we speed up the CPU by a factor of 10, while neglecting I/O. Amdahl's Law tells us that we will get a speedup of only 5 times, with half the potential of the CPU wasted. Similarly, making the CPU 100 times faster without improving the I/O would obtain a speedup of only 10 times, squandering 90% of the potential. If, as predicted in Chapter 1, performance of CPUs improves at 55% per year and I/O does not improve, every task will become I/O-bound. There would be no reason to buy faster CPUs—and no jobs for CPU designers.

To reflect the increasing importance of I/O, we have expanded its coverage in this second edition. We now have two I/O chapters: this chapter covers storage I/O, and the next covers network I/O. Although two chapters cannot fully vindicate I/O, they may at least atone for some of the sins of the past and restore some balance.

Are CPUs Ever Idle?

Some suggest that the prejudice against I/O is well founded. I/O speed doesn't matter, they argue, since there is always another process to run while one process waits for a peripheral.

There are several points to make in reply. First, this is an argument that performance is measured as *throughput*—more tasks per hour—rather than as response time. Plainly, if users didn't care about response time, interactive software never would have been invented, and there would be no workstations or personal computers today; section 6.4 gives experimental evidence on the importance of response time. It may also be expensive to rely on performing other processes while waiting for I/O, since the main memory must be large or else the paging traffic from process switching would actually increase I/O. Furthermore, with desktop computing there is only one person per CPU, and thus fewer processes than in timesharing; many times the only waiting process is the human being! And some applications, such as transaction processing (section 6.4), place strict limits on response time as part of the performance analysis.

Thus, I/O performance can limit system performance and effectiveness.

6.2 Types of Storage Devices

Rather than discuss the characteristics of all storage devices, we will concentrate on the devices with the highest capacity: magnetic disks, magnetic tapes, CD-ROMS, and automated tape libraries.

Magnetic Disks

I think Silicon Valley was misnamed. If you look back at the dollars shipped in products in the last decade there has been more revenue from magnetic disks than from silicon. They ought to rename the place Iron Oxide Valley.

Al Hoagland, One of the Pioneers of Magnetic Disks (1982)

In spite of repeated attacks by new technologies, magnetic disks have dominated secondary storage since 1965. Magnetic disks play two roles in computer systems:

- Long-term, nonvolatile storage for files, even when no programs are running
- A level of the memory hierarchy below main memory used for virtual memory during program execution (see section 5.7)

In this chapter we are not talking about floppy disks, but the original "hard" disks.

As descriptions of magnetic disks can be found in countless books, we will only list the key characteristics, with the terms illustrated in Figure 6.1. A magnetic disk consists of a collection of *platters* (1 to 20), rotating on a spindle at, say, 3600 revolutions per minute (RPM). These platters are metal disks covered with magnetic recording material on both sides. Disk diameters vary by a factor of six, from 1.3 to 8 inches. Traditionally, the widest disks have the highest performance and the smallest disks have the lowest cost per disk drive.

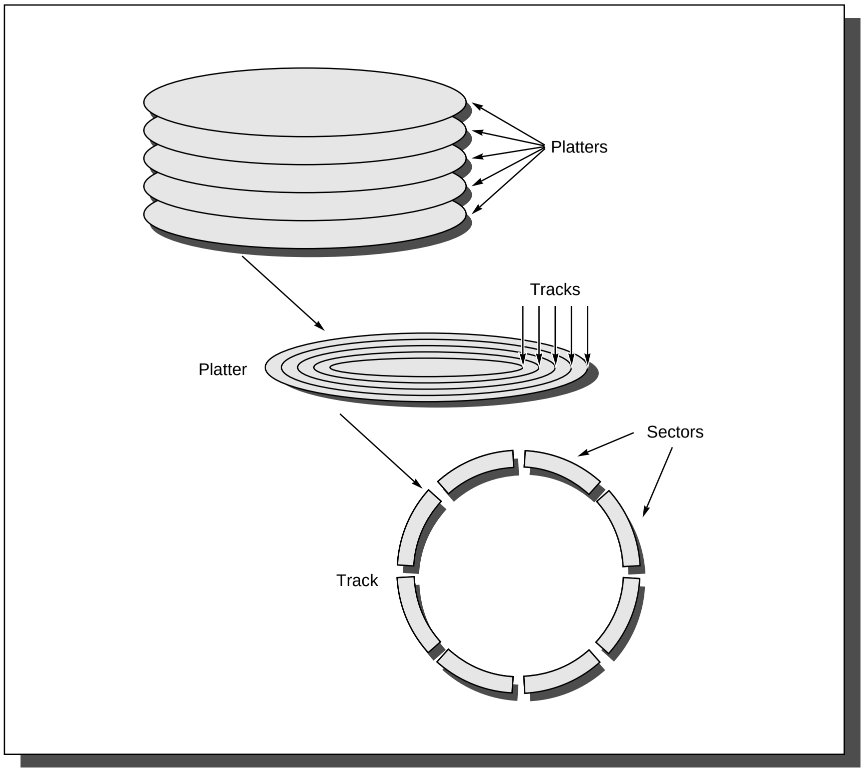


FIGURE 6.1 Disks are organized into platters, tracks, and sectors. Both sides of a platter are coated so that information can be stored on both surfaces. A cylinder refers to a track at the same position on every platter.

The disk surface is divided into concentric circles, designated *tracks*. There are typically 500 to 2500 tracks on each surface. Each track in turn is divided into *sectors* that contain the information; each track might have 64 sectors. A sector is the smallest unit that can be read or written. The sequence recorded on the magnetic media is a sector number, a gap, the information for that sector including error correction code, a gap, the sector number of the next sector, and so on.

Traditionally, all tracks have the same number of sectors; the outer tracks, which are longer, record information at a lower density than the inner tracks. Recording more sectors on the outer tracks than on the inner tracks, called *constant bit density,* is becoming more widespread with the advent of intelligent interface standards such as SCSI (see section 6.3). IBM mainframes allow users to select the size of the sectors, although almost all other systems fix their size.

To read and write information into a sector, a movable *arm* containing a *read/write head* is located over each surface. Bits are recorded using a run-length limited code, which improves the recording density of the magnetic media. The arms for each surface are connected together and move in conjunction, so that every arm is over the same track of every surface. The term *cylinder* is used to refer to all the tracks under the arms at a given point on all surfaces.

To read or write a sector, the disk controller sends a command to move the arm over the proper track. This operation is called a *seek*, and the time to move the arm to the desired track is called *seek time*. Average seek time is the subject of considerable misunderstanding.

Disk manufacturers report minimum seek time, maximum seek time, and average seek time in their manuals. The first two are easy to measure, but the average was open to wide interpretation. The industry decided to calculate average seek time as the sum of the time for all possible seeks divided by the number of possible seeks. Average seek times are advertised to be 8 ms to 12 ms, but, depending on the application and operating system, the actual average seek time may be only 25% to 33% of the advertised number, due to locality of disk references. Section 6.9 has a detailed example.

The time for the requested sector to rotate under the head is the *rotation latency* or *rotational delay.* Many disks rotate at 3600 RPM, and an average latency to the desired information is halfway around the disk; the average rotation time for many disks is therefore

$$\text{Average rotation time} = \frac{0.5}{3600\ \text{RPM}} = 0.0083\ \text{sec} = 8.3\ \text{ms}$$

Note that there are two mechanical components to a disk access: several milliseconds on average for the arm to move over the desired track and then several milliseconds on average for the desired sector to rotate under the read/write head.

The next component of disk access, *transfer time,* is the time it takes to transfer a block of bits, typically a sector, under the read-write head. This time is a function of the block size, rotation speed, recording density of a track, and speed of the electronics connecting disk to computer. Transfer rates in 1995 are typically 2 to 8 MB per second.

Between the disk controller and main memory is a hierarchy of controllers and data paths, whose complexity varies (and the cost of the computer with it). For example, whenever the transfer time is a small portion of a full access, the designer will want to disconnect the memory device during the access so that others can transfer their data. This is true for disk controllers in high-performance systems, and, as we shall see later, for buses and networks. There is also a desire to amortize this long access by reading more than simply what is requested; this is called *read ahead*. The hope is that a nearby request will be for the next sector, which will already be available.

To handle the complexities of disconnect/connect and read ahead, there is usually, in addition to the disk drive, a device called a *disk controller*. Thus, the final component of disk-access time is *controller time*, which is the overhead the controller imposes in performing an I/O access. When referring to the performance of a disk in a computer system, the time spent waiting for a disk to become free (*queuing delay*) is added to this time.

EXAMPLE What is the average time to read or write a 512-byte sector for a typical disk? The advertised average seek time is 9 ms, the transfer rate is 4 MB/sec, it rotates at 7200 RPM, and the controller overhead is 1 ms. Assume the disk is idle so that there is no queuing delay.

ANSWER Average disk access is equal to average seek time + average rotational delay + transfer time + controller overhead. Using the calculated, average seek time, the answer is

$$9 \text{ ms} + \frac{0.5}{7200 \text{ RPM}} + \frac{0.5 \text{ KB}}{4.0 \text{ MB/sec}} + 1 \text{ ms} = 9 + 4.15 + 0.125 + 1 = 14.3 \text{ ms}$$

Assuming the measured seek time is 33% of the calculated average, the answer is

$$3 \text{ ms} + 4.2 \text{ ms} + 0.1 \text{ ms} + 1 \text{ ms} = 8.3 \text{ ms}$$

■

Figure 6.2 shows the characteristics of a 1993 magnetic disk. Large-diameter drives have many more megabytes to amortize the cost of electronics, so the traditional wisdom used to be that they had the lowest cost per megabyte. But this advantage is offset for the small drives by the much higher sales volume, which lowers manufacturing costs. Hence the best price per megabyte is typically a medium-width disk, which has enough capacity to offset the cost of the mechanical components and enough volume to take advantage of high manufacturing volume.

Characteristics	**Seagate ST31401N Elite-2 SCSI Drive**
Disk diameter (inches)	5.25
Formatted data capacity (GB)	2.8
Cylinders	2627
Tracks per cylinder	21
Sectors per track	≈ 99
Bytes per sector	512
Rotation speed (RPM)	5400
Average seek in ms (random cylinder to cylinder)	11.0
Minimum seek in ms	1.7
Maximum seek in ms	22.5
Data transfer rate in MB/sec	≈ 4.6

FIGURE 6.2 Characteristics of a 1993 magnetic disk.

The Future of Magnetic Disks

The disk industry has concentrated on improving the capacity of disks. Improvement in capacity is customarily expressed as *areal density*, measured in bits per square inch:

$$\text{Areal density} = \frac{\text{Tracks}}{\text{Inch}} \text{ on a disk surface} \times \frac{\text{Bits}}{\text{Inch}} \text{ on a track}$$

Through about 1988 the rate of improvement of areal density was 29% per year, thus doubling density every three years. Since that time the rate has improved to 60% per year, quadrupling density every three years and matching the traditional rate of DRAMs. In 1995 the highest density in commercial products is 644 million bits per square inch, with 3000 million bits per square inch demonstrated in the labs.

Cost per megabyte has dropped at least at the same rate of improvement of areal density, with smaller drives playing the larger role in this improvement. Figure 6.3 plots price per personal computer disk between 1983 and 1995, showing both the rapid drop in price and the increase in capacity. Figure 6.4 translates these costs into price per megabyte, showing that it has improved more than a hundredfold over those 12 years. In fact, between 1992 and 1995 the rate of improvement in cost per megabyte of personal computer disks was about 2.0 times per year, a considerable increase over the previous rate of about 1.3 to 1.4 times per year between 1986 and 1992.

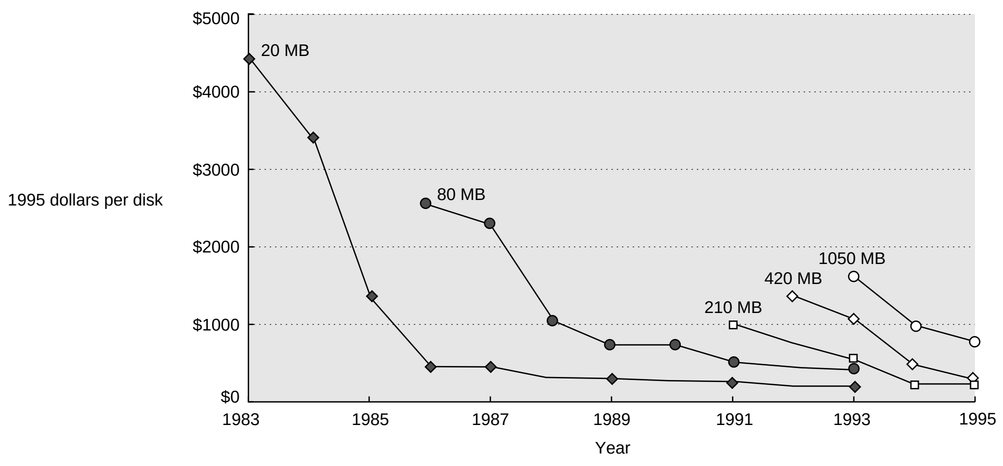


FIGURE 6.3 Price per personal computer disk over time. The prices are in 1995 dollars, adjusted for inflation using the Producer Price Index. The costs were collected from advertisements from the January edition of *Byte* magazine, using the lowest price of a disk of a particular size in that issue. In a few cases, the price was adjusted slightly to get a consistent disk capacity (e.g., shrinking the price of an 86-MB disk by 80/86 to get a point for the 80-MB line).

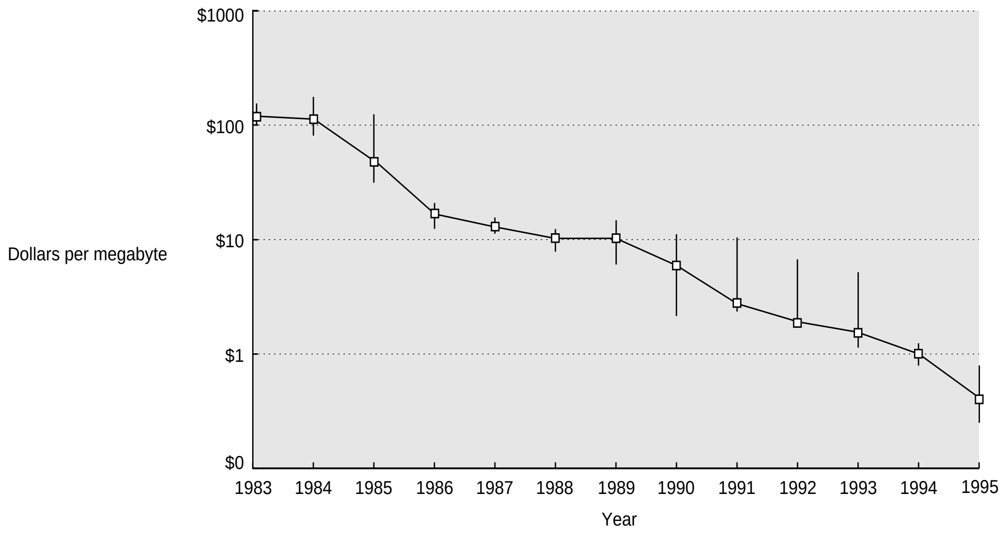


FIGURE 6.4 Price per megabyte of personal computer disk over time. The center point is the median price per MB, with the low point on the line being the minimum and the high point being the maximum. These data were collected in the same way as for Figure 6.3, except that more disks are included on this graph.

Because it is easier to spin the smaller mass, smaller-diameter disks save power as well as volume. Smaller drives also have fewer cylinders, so the seek distances are shorter. In 1995, 3.5-inch or 2.5-inch drives are probably the leading technology, and the future will see even smaller drives. Increasing density (bits per inch on a track) has improved transfer times, and there has been some small improvement in seek speed. Rotation speeds have improved from the standard 3600 RPM in the 1980s to 5400–7200 RPM in the 1990s.

Magnetic disks have been challenged many times for supremacy of secondary storage. One reason has been the fabled *access time gap* as shown in Figure 6.5. The price of a megabyte of disk storage in 1995 is about 100 times cheaper than the price of a megabyte of DRAM in a system, but DRAM is about 100,000 times faster. Many a scientist has tried to invent a technology to fill that gap, but thus far all have failed.

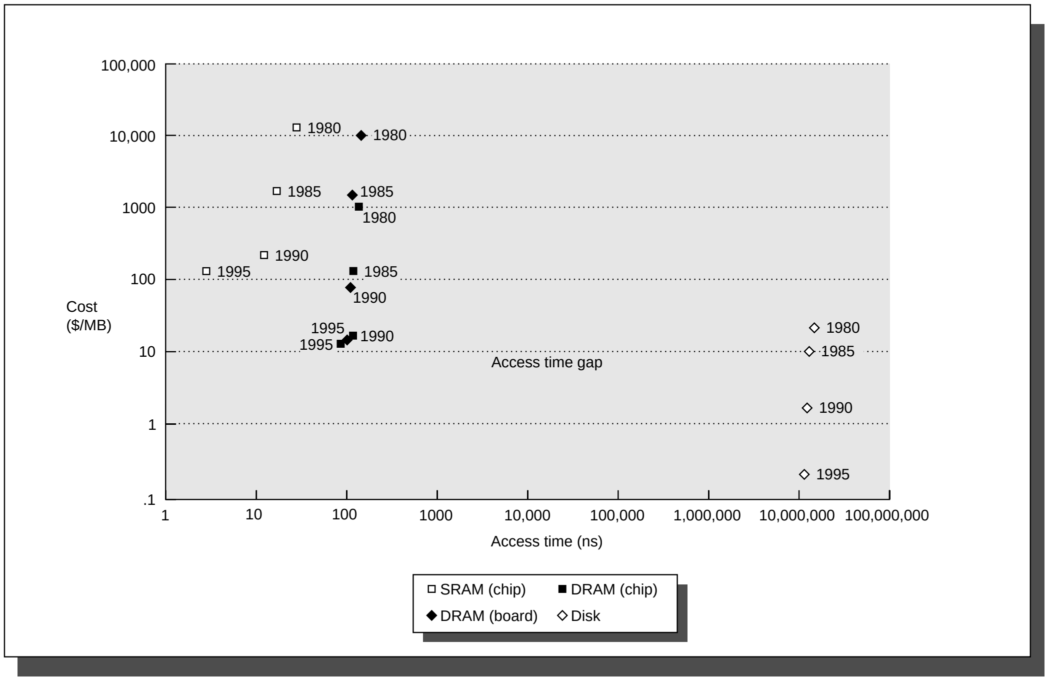


FIGURE 6.5 Cost versus access time for SRAM, DRAM, and magnetic disk in 1980, 1985, 1990, and 1995. (Note the difference in cost between a DRAM chip and DRAM chips packaged on a board and ready to plug into a computer.) The two-order-of-magnitude gap in cost and access times between semiconductor memory and rotating magnetic disks has inspired a host of competing technologies to try to fill it. So far, such attempts have been made obsolete before production by improvements in magnetic disks, DRAMs, or both. Note that between 1990 and 1995 the cost per megabyte of SRAM and DRAM chips made less improvement, while disk cost made dramatic improvement. Also, since 1990 SIMM modules have shrunk the gap between the cost of DRAM (board) and DRAM (chip).

Using DRAMs as Disks

One challenger to disks for dominance of secondary storage is *solid state disks* (SSDs), built from DRAMs with a battery to make the system nonvolatile; another is *expanded storage* (ES), a large memory that allows only block transfers to or from main memory. ES acts like a software-controlled cache (the CPU stalls during the block transfer), while SSDs involve the operating system just like a transfer from magnetic disks. The advantages of SSDs and ES are nonvolatility, trivial seek times, higher potential transfer rate, and possibly higher reliability. Unlike just a larger main memory, SSDs and ES are autonomous: They require special commands to access their storage, and thus are "safe" from some software errors that write over main memory. The block-access nature of SSDs and ES allows error correction to be spread over more words, which means lower cost for greater error recovery. For example, IBM's ES uses the greater error recovery to allow it to be constructed from less reliable (and less expensive) DRAMs without sacrificing product availability. SSDs, unlike main memory and ES, may be shared by multiple CPUs because they function as separate units. Placing DRAMs in an I/O device rather than memory is also one way to get around the address-space limits of the current 32-bit computers. The disadvantage of SSDs and ES is cost, which is at least 50 times per megabyte the cost of magnetic disks.

When the first edition of this book was written, disks were growing at 29% per year and DRAMs at 60% per year. One exercise asked when DRAMs would match the cost per bit of magnetic disks. Now that disks have at least matched the DRAM growth rate and will apparently do so for many years, the question has changed from What year? to What must change for it to happen?

Optical Disks

Another challenger to magnetic disks is *optical compact disks*, or *CDs*. The *CD-ROM* is removable and inexpensive to manufacture, but it is a read-only medium. Its low manufacturing cost has made it a favorite medium for distributing information, but not as a rewritable storage device. The high capacity and low cost mean that CD-ROMs may well replace floppy disks as the favorite medium for distributing personal computer software.

So far, magnetic disk challengers have never had a product to market at the right time. By the time a new product ships, disks have made advances as predicted earlier, and costs have dropped accordingly.

Unfortunately, the data distribution responsibilities of CDs mean that their rate of improvement is governed by standards committees, and it appears that magnetic storage grows more quickly than human beings can agree on CD standards. Writable optical disks, however, may have the potential to compete with new tape technologies for archival storage.

Magnetic Tapes

Magnetic tapes have been part of computer systems as long as disks because they use the same technology as disks, and hence follow the same density improvements. The inherent cost/performance difference between disks and tapes is based on their geometries:

- Fixed rotating platters offer random access in milliseconds, but disks have a limited storage area and the storage medium is sealed within each reader.
- Long strips wound on removable spools of "unlimited" length mean many tapes can be used per reader, but tapes require sequential access that can take seconds.

This relationship has made tapes the technology of choice for backups to disk.

One of the limits of tapes has been the speed at which the tapes can spin without breaking or jamming. A relatively recent technology, called *helical scan tapes*, solves this problem by keeping the tape speed the same but recording the information on a diagonal to the tape with a tape reader that spins much faster than the tape is moving. This technology increases recording density by about a factor of 20 to 50. Helical scan tapes were developed for the low-cost VCRs and camcorders, which brings down the cost of the tapes and readers.

One drawback to tapes is that they wear out: Helical tapes last for hundreds of passes, while the traditional longitudinal tapes wear out in thousands to millions of passes. The helical scan read/write heads also wear out quickly, typically rated for 2000 hours of continuous use. Finally, there are typically long rewind, eject, load, and spin-up times for helical scan tapes. In the archival backup market, such performance characteristics have not mattered, and hence there has been more engineering focus on increasing density than on overcoming these limitations.

Automated Tape Libraries

Tape capacities are enhanced by inexpensive robots to automatically load and store tapes, offering a new level of storage hierarchy. These robo-line tapes mean access to terabytes of information in tens of seconds, without the intervention of a human operator. Figure 6.6 shows the Storage Technologies Corporation (STC) PowderHorn, which loads 6000 tapes, giving a total capacity of 60 terabytes. Putting this capacity into perspective, in 1995 the Library of Congress is estimated to have 30 terabytes of text, if books could be magically transformed into ASCII characters.

One interesting characteristic of automated tape libraries is that economy of scale can apply, unlike almost all other parts of the computer industry. Figure 6.7 shows that the price per gigabyte drops by a factor of four when going from the small systems (less than 100 GB in 1995) to the large systems (greater than 1000 GB). The drawback of such large systems is the limited bandwidth of this massive storage.

Now that we have described several storage devices, we must discover how to connect them to a computer.

FIGURE 6.6 The STC PowderHorn. This storage silo holds 6000 tape cartridges; using the 3590 cartridge announced in 1995, the total capacity is 60 terabytes. It has a performance level of up to 350 cartridge exchanges per hour. (Courtesy STC.)

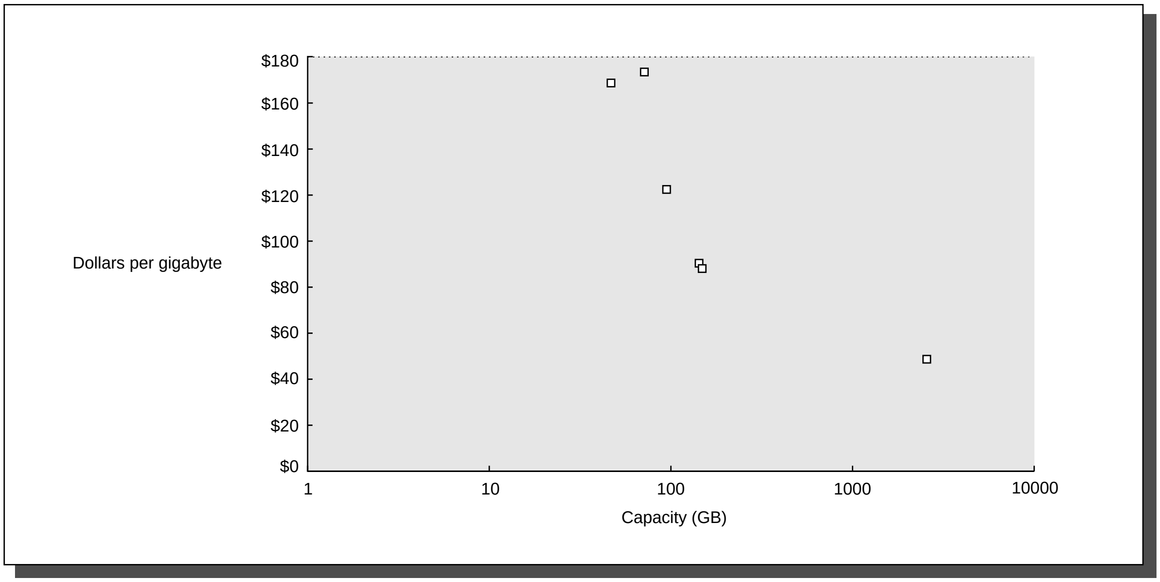


FIGURE 6.7 Plot of capacity per library versus dollars per gigabyte for several 1995 tape libraries. Note that the x axis is a log scale. In 1995 large libraries are one-quarter the cost per gigabyte of small libraries. The danger of comparing disk to tape at small capacities is the subject of the fallacy discussed on page 548.

6.3 Buses—Connecting I/O Devices to CPU/Memory

In a computer system, the various subsystems must have interfaces to one another; for instance, the memory and CPU need to communicate, and so do the CPU and I/O devices. This is commonly done with a *bus*. The bus serves as a shared communication link between the subsystems. The two major advantages of the bus organization are low cost and versatility. By defining a single interconnection scheme, new devices can be added easily and peripherals may even be moved between computer systems that use a common bus. The cost is low, since a single set of wires is shared in multiple ways.

The major disadvantage of a bus is that it creates a communication bottleneck, possibly limiting the maximum I/O throughput. When I/O must pass through a central bus, this bandwidth limitation is as real as—and sometimes more severe than—memory bandwidth. In commercial systems, where I/O is frequent, and in supercomputers, where the necessary I/O rates are high because the CPU performance is high, designing a bus system capable of meeting the demands of the processor is a major challenge.

One reason bus design is so difficult is that the maximum bus speed is largely limited by physical factors: the length of the bus and the number of devices (and, hence, bus loading). These physical limits prevent arbitrary bus speedup. The desire for high I/O rates (low latency) and high I/O throughput can also lead to conflicting design requirements.

Buses are traditionally classified as *CPU-memory buses* or *I/O buses*. I/O buses may be lengthy, may have many types of devices connected to them, have a wide range in the data bandwidth of the devices connected to them, and normally follow a bus standard. CPU-memory buses, on the other hand, are short, generally high speed, and matched to the memory system to maximize memory-CPU bandwidth. During the design phase, the designer of a CPU-memory bus knows all the types of devices that must connect together, while the I/O bus designer must accept devices varying in latency and bandwidth capabilities. To lower costs, some computers have a single bus for both memory and I/O devices.

Let's review a typical *bus transaction,* as seen in Figure 6.8. A bus transaction includes two parts: sending the address and receiving or sending the data. Bus transactions are usually defined by what they do to memory: A *read* transaction transfers data *from* memory (to either the CPU or an I/O device), and a *write* transaction writes data to the memory. In a read transaction, the address is first sent down the bus to the memory, together with the appropriate control signals indicating a read. In Figure 6.8, this means deasserting the read signal. The memory responds by returning the data on the bus with the appropriate control signals, in this case deasserting the wait signal. A write transaction requires that the CPU or I/O device send both address and data and requires no return of data. Usually the CPU must wait between sending the address and receiving the data on a read, but the CPU often does not wait on writes.

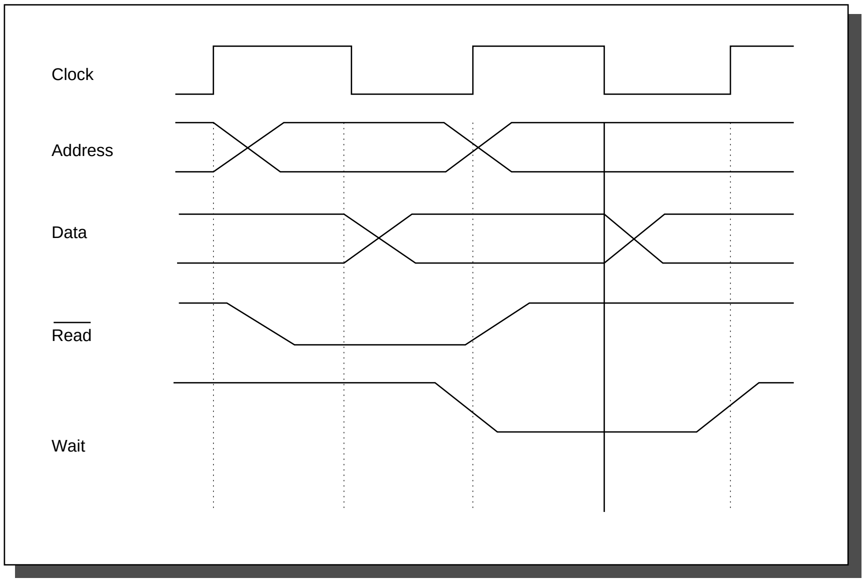


FIGURE 6.8 Typical bus read transaction. This bus is synchronous. The read begins when the read signal is asserted, and data are not ready until the wait signal is deasserted.

Bus Design Decisions

The design of a bus presents several options, as Figure 6.9 shows. Like the rest of the computer system, decisions will depend on cost and performance goals. The first three options in the figure are clear choices—separate address and data lines, wider data lines, and multiple-word transfers all give higher performance at more cost.

Option	High performance	Low cost
Bus width	Separate address and data lines	Multiplex address and data lines
Data width	Wider is faster (e.g., 64 bits)	Narrower is cheaper (e.g., 8 bits)
Transfer size	Multiple words have less bus overhead	Single-word transfer is simpler
Bus masters	Multiple (requires arbitration)	Single master (no arbitration)
Split transaction?	Yes—separate request and reply packets get higher bandwidth (need multiple masters)	No—continuous connection is cheaper and has lower latency
Clocking	Synchronous	Asynchronous

FIGURE 6.9 The main options for a bus. The advantage of separate address and data buses is primarily on writes.

The next item in the table concerns the number of *bus masters*. These are devices that can initiate a read or write transaction; the CPU, for instance, is always a bus master. A bus has multiple masters when there are multiple CPUs or when I/O devices can initiate a bus transaction. If there are multiple masters, an arbitration scheme is required among the masters to decide who gets the bus next. Arbitration is often a fixed priority, as is the case with daisy-chained devices or an approximately fair scheme that randomly chooses which master gets the bus.

With multiple masters, a bus can offer higher bandwidth by going to packets, as opposed to holding the bus for the full transaction. This technique is called *split transactions*. (Some systems call this ability *connect/disconnect*, a *pipelined bus,* or a *packet-switched bus*.) Figure 6.10 shows the split-transaction bus. The read transaction is broken into a read-request transaction that contains the address and a memory-reply transaction that contains the data. Each transaction must now be tagged so that the CPU and memory can tell what is what. Split transactions make the bus available for other masters while the memory reads the words from the requested address. It also normally means that the CPU must arbitrate for the bus to send the data and the memory must arbitrate for the bus to return the data. Thus, a split-transaction bus has higher bandwidth, but it usually has higher latency than a bus that is held during the complete transaction.

The final item in Figure 6.9, *clocking,* concerns whether a bus is synchronous or asynchronous. If a bus is *synchronous,* it includes a clock in the control lines and a fixed protocol for address and data relative to the clock. Since little or no logic is needed to decide what to do next, these buses can be both fast and

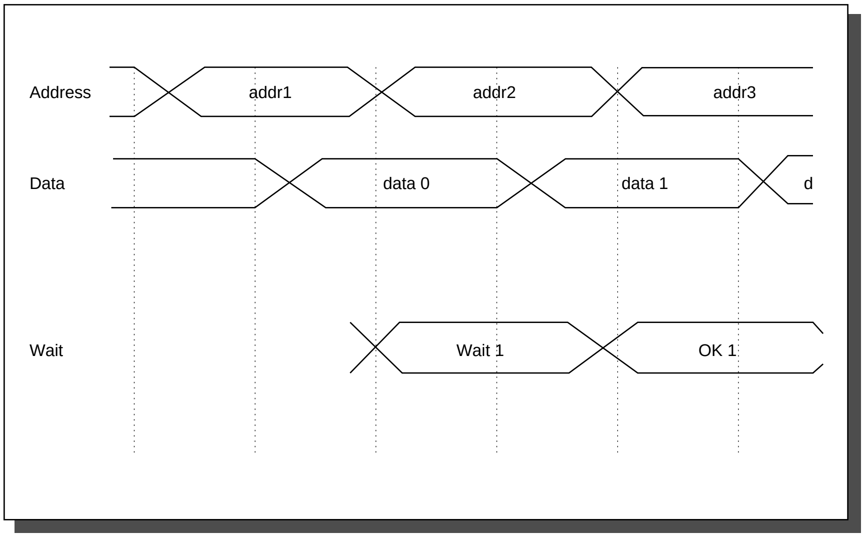


FIGURE 6.10 A split-transaction bus. Here the address on the bus corresponds to a later memory access.

inexpensive. They have two major disadvantages, however. Everything on the bus must run at the same clock rate, and because of clock-skew problems, synchronous buses cannot be long. CPU-memory buses are typically synchronous.

An *asynchronous* bus, on the other hand, is not clocked. Instead, self-timed, handshaking protocols are used between bus sender and receiver. Figure 6.11 shows the steps of a master performing a write on an asynchronous bus.

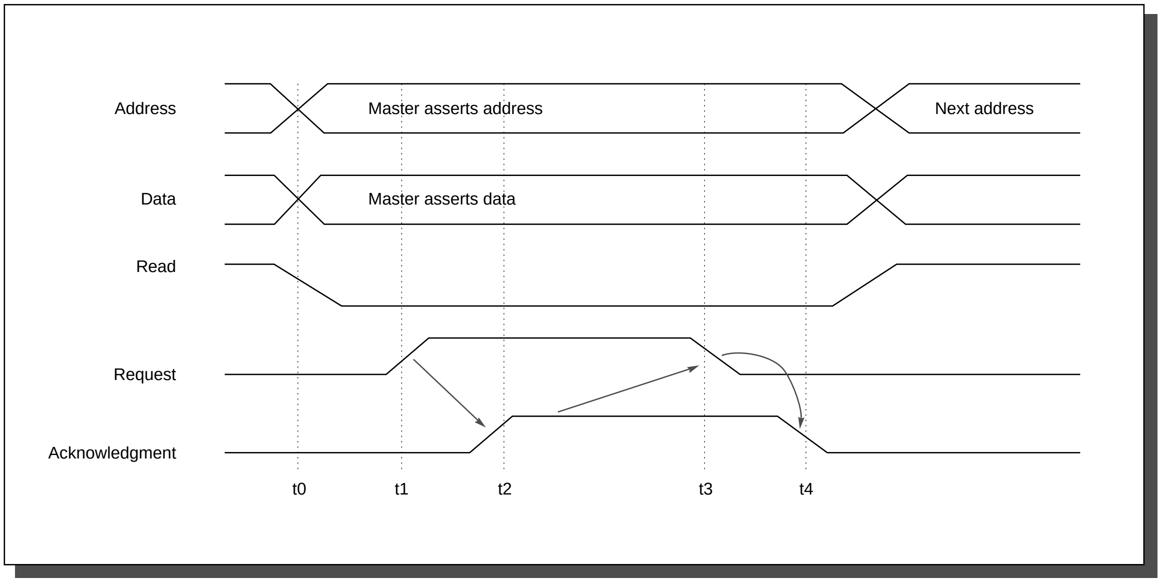


FIGURE 6.11 A master performing a write. The state of the transaction at each time step is as follows. Master has obtained control and asserts address, direction, and data, then waits a specified amount of time for slaves to decode target; t1: Master asserts request line; t2: Slave asserts ack, indicating data received; t3: Master releases req; t4: Slave releases ack.

Asynchrony makes it much easier to accommodate a wide variety of devices and to lengthen the bus without worrying about clock skew or synchronization problems. If a synchronous bus can be used, it is usually faster than an asynchronous bus because it avoids the overhead of synchronizing the bus for each transaction. The choice of synchronous versus asynchronous bus has implications not only for data bandwidth but also for an I/O system's capacity in terms of physical distance and number of devices that can be connected to the bus; asynchronous buses scale better with technological changes. I/O buses are typically asynchronous. Figure 6.12 suggests when to use one over the other.

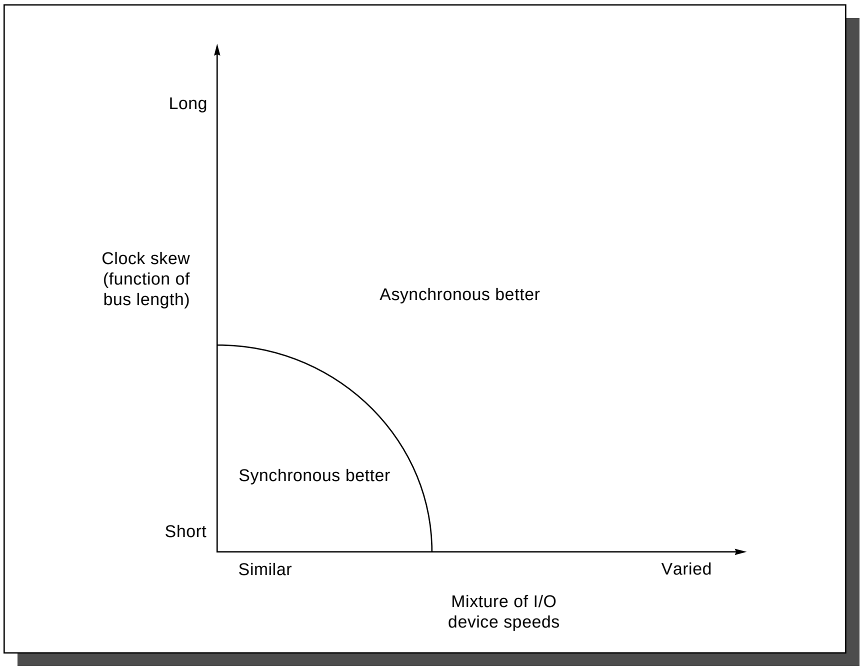

Long
Clock skew (function of bus length)
Asynchronous better
Synchronous better
Short
Similar
Varied
Mixture of I/O device speeds

FIGURE 6.12 Preferred bus type as a function of length/clock skew and variation in I/O device speed. Synchronous is best when the distance is short and the I/O devices on the bus all transfer at similar speeds.

Bus Standards

The number and variety of I/O devices are not fixed on most computer systems, permitting customers to tailor computers to their needs. As the interface to which devices are connected, the I/O bus can also be considered an expansion bus for adding I/O devices over time. Standards that let the computer designer and I/O-device designer work independently, therefore, play a large role in determining the choice of buses. As long as both the computer-system designer and the I/O-device designer meet the requirements, any I/O device can connect to any computer. In fact, an I/O bus standard is the document that defines how to connect them.

Machines sometimes grow to be so popular that their I/O buses become de facto standards; examples are the PDP-11 Unibus and the IBM PC-AT Bus. Once many I/O devices have been built for a popular machine, other computer designers will build their I/O interface so that those devices can plug into their machines as well. Sometimes standards also come from an explicit standards effort on the part of I/O device makers. The *intelligent peripheral interface* (IPI) and Ethernet are examples of standards that resulted from the cooperation of manufacturers. If standards are successful, they are eventually blessed by a sanctioning body like ANSI or IEEE. Occasionally, a bus standard comes top-down directly from a standards committee—PCI is one example.

Examples of Buses

Figure 6.13 summarizes characteristics of five I/O buses, and Figure 6.14 summarizes three CPU-memory buses found in servers.

	S bus	**MicroChannel**	**PCI**	**IPI**	**SCSI 2**
Data width (primary)	32 bits	32 bits	32 to 64 bits	16 bits	8 to 16 bits
Clock rate	16 to 25 MHz	Asynchronous	33 MHz	Asynchronous	10 MHz or asynchronous
Number of bus masters	Multiple	Multiple	Multiple	Single	Multiple
Bandwidth, 32-bit reads	33 MB/sec	20 MB/sec	33 MB/sec	25 MB/sec	20 MB/sec or 6 MB/sec
Bandwidth, peak	89 MB/sec	75 MB/sec	132 MB/sec	25 MB/sec	20 MB/sec or 6 MB/sec
Standard	None	—	—	ANSI X3.129	ANSI X3.131

FIGURE 6.13 Summary of I/O buses. The first two started as CPU-memory buses and evolved into I/O buses.

	HP Summit	**SGI Challenge**	**Sun XDBus**
Data width (primary)	128 bits	256 bits	144 bits
Clock rate	60 MHz	48 MHz	66 MHz
Number of bus masters	Multiple	Multiple	Multiple
Bandwidth, peak	960 MB/sec	1200 MB/sec	1056 MB/sec
Standard	None	None	None

FIGURE 6.14 Summary of CPU-memory buses found in 1994 servers. All use split transactions to enhance bandwidth. The number of slots on the bus for masters or slaves is 16, 9, and 10, respectively.

Interfacing Storage Devices to the CPU

Having described I/O devices and looked at some of the issues of the connecting bus, we are ready to discuss the CPU end of the interface. The first question is how the physical connection of the I/O bus should be made. The two choices are connecting it to memory or to the cache. In the following section we will discuss the pros and cons of connecting an I/O bus directly to the cache; in this section we examine the more usual case in which the I/O bus is connected to the main memory bus. Figure 6.15 shows a typical organization. In low-cost systems, the I/O bus *is* the memory bus; this means an I/O command on the bus could interfere with a CPU instruction fetch, for example.

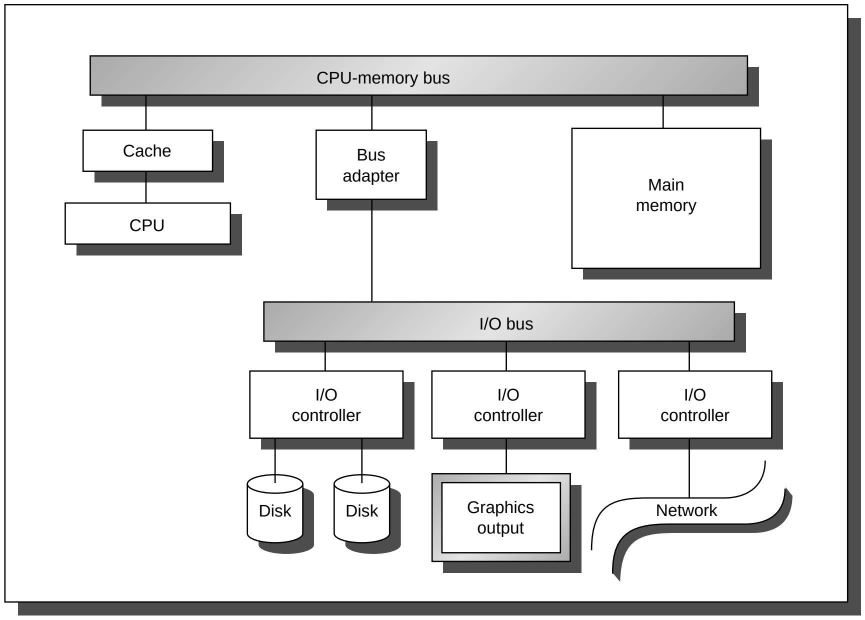


FIGURE 6.15 A typical interface of I/O devices and an I/O bus to the CPU-memory bus.

Once the physical interface is chosen, the question becomes, How does the CPU address an I/O device that it needs to send or receive data? The most common practice is called *memory-mapped* I/O. In this scheme, portions of the address space are assigned to I/O devices. Reads and writes to those addresses may cause data to be transferred; some portion of the I/O space may also be set aside for device control, so commands to the device are just accesses to those memory-mapped addresses.

The alternative practice is to use dedicated I/O opcodes in the CPU. In this case, the CPU sends a signal that this address is for I/O devices. Examples of computers with I/O instructions are the Intel 80x86 and the IBM 370 computers. I/O opcodes have been waning in popularity.

No matter which addressing scheme is selected, each I/O device has registers to provide status and control information. Either through loads and stores in memory-mapped I/O or through special instructions, the CPU sets flags to determine the operation the I/O device will perform.

Any I/O event is rarely a single operation. For example, the DEC LP11 line printer has two I/O device registers: one for status information and one for data to be printed. The status register contains a *done bit*, set by the printer when it has printed a character, and an *error bit*, indicating that the printer is jammed or out of paper. Each byte of data to be printed is put into the data register; the CPU must then wait until the printer sets the done bit before it can place another character in the buffer.

This simple interface, in which the CPU periodically checks status bits to see if it is time for the next I/O operation, is called *polling*. As you might expect, the fact that CPUs are so much faster than I/O devices means that polling may waste a lot of CPU time. This was recognized long ago, leading to the invention of interrupts to notify the CPU when it is time to do something for the I/O device.

Interrupt-driven I/O, used by most systems for at least some devices, allows the CPU to work on some other process while waiting for the I/O device. For example, the LP11 has a mode that allows it to interrupt the CPU whenever the done bit or error bit is set. In general-purpose applications, interrupt-driven I/O is the key to multitasking operating systems and good response times.

The drawback to interrupts is the operating system overhead on each event. In real-time applications with hundreds of I/O events per second, this overhead can be intolerable. One hybrid solution for real-time systems is to use a clock to periodically interrupt the CPU, at which time the CPU polls all I/O devices.

Delegating I/O Responsibility from the CPU

Interrupt-driven I/O relieves the CPU from waiting for every I/O event, but there are still many CPU cycles spent in transferring data. Transferring a disk block of 2048 words, for instance, would require at least 2048 loads and 2048 stores, as well as the overhead for the interrupt. Since I/O events so often involve block transfers, *direct memory access* (DMA) hardware is added to many computer systems to allow transfers of numbers of words without intervention by the CPU.

DMA is a specialized processor that transfers data between memory and an I/O device while the CPU goes on with other tasks. Thus, it is external to the CPU and must act as a master on the bus. The CPU first sets up the DMA registers, which contain a memory address and number of bytes to be transferred. Once the DMA transfer is complete, the controller interrupts the CPU. There may be multiple DMA devices in a computer system; for example, DMA is frequently part of the controller for an I/O device.

Increasing the intelligence of the DMA device can further unburden the CPU. Devices called *I/O processors* (or *I/O controllers*, or *channel controllers*) operate either from fixed programs or from programs downloaded by the operating system. The operating system typically sets up a queue of *I/O control blocks* that contain information such as data location (source and destination) and data size. The I/O processor then takes items from the queue, doing everything requested and sending a single interrupt when the task specified in the I/O control blocks is complete. Whereas the LP11 line printer would cause 4800 interrupts to print a 60-line by 80-character page, an I/O processor could save 4799 of those interrupts.

I/O processors are similar to multiprocessors in that they facilitate several processes being executed simultaneously in the computer system. I/O processors are less general than CPUs, however, since they have dedicated tasks, and thus parallelism is also much more limited. Also, an I/O processor doesn't normally change information, as a CPU does, but just moves information from one place to another.

Now that we have covered the basic types of storage devices and ways to connect them to the CPU, we are ready to look at ways to evaluate the performance of storage systems.

6.4 I/O Performance Measures

I/O performance has measures that have no counterparts in CPU design. One of these is diversity: Which I/O devices can connect to the computer system? Another is capacity: How many I/O devices can connect to a computer system?

In addition to these unique measures, the traditional measures of performance, response time and throughput, also apply to I/O. (I/O throughput is sometimes called *I/O bandwidth*, and response time is sometimes called *latency.*) The next two figures offer insight into how response time and throughput trade off against each other. Figure 6.16 shows the simple producer-server model. The producer creates tasks to be performed and places them in a buffer; the server takes tasks from the first-in-first-out buffer and performs them.

Response time is defined as the time a task takes from the moment it is placed in the buffer until the server finishes the task. Throughput is simply the average number of tasks completed by the server over a time period. To get the highest possible throughput, the server should never be idle, and thus the buffer should never be empty. Response time, on the other hand, counts time spent in the buffer and is therefore minimized by the buffer being empty.

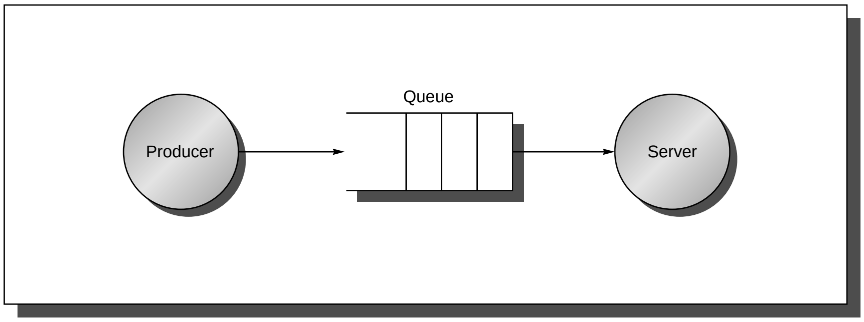


FIGURE 6.16 The traditional producer-server model of response time and throughput. Response time begins when a task is placed in the buffer and ends when it is completed by the server. Throughput is the number of tasks completed by the server in unit time.

Another measure of I/O performance is the interference of I/O with CPU execution. Transferring data may interfere with the execution of another process. There is also overhead due to handling I/O interrupts. Our concern here is how many more clock cycles a process will take because of I/O for another process.

Throughput versus Response Time

Figure 6.17 shows throughput versus response time (or latency) for a typical I/O system. The knee of the curve is the area where a little more throughput results in much longer response time or, conversely, a little shorter response time results in much lower throughput.

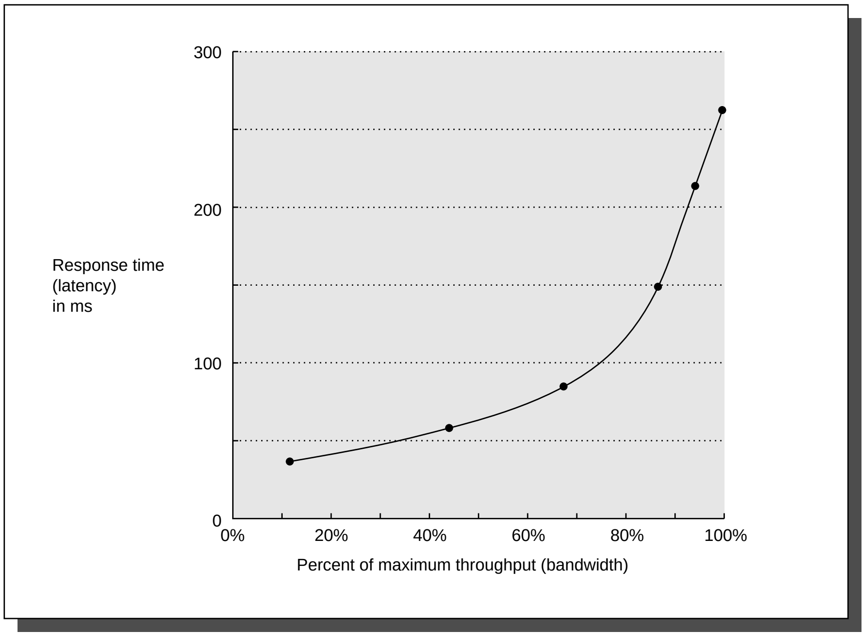


FIGURE 6.17 Throughput versus response time. Latency is normally reported as response time. Note that the minimum response time achieves only 11% of the throughput, while the response time for 100% throughput takes seven times the minimum response time. Note that the independent variable in this curve is implicit: To trace the curve, you typically vary load (concurrency). Chen et al. [1990] collected these data for an array of magnetic disks.

Life would be simpler if improving performance always meant improvements in both response time and throughput. Adding more servers, as in Figure 6.18, increases throughput: By spreading data across two disks instead of one, tasks may be serviced in parallel. Alas, this does not help response time, unless the workload is held constant and the time in the buffers is reduced because of more resources.

How does the architect balance these conflicting demands? If the computer is interacting with human beings, Figure 6.19 suggests an answer. This figure presents the results of two studies of interactive environments: one keyboard oriented and one graphical. An interaction, or *transaction,* with a computer is divided into three parts:

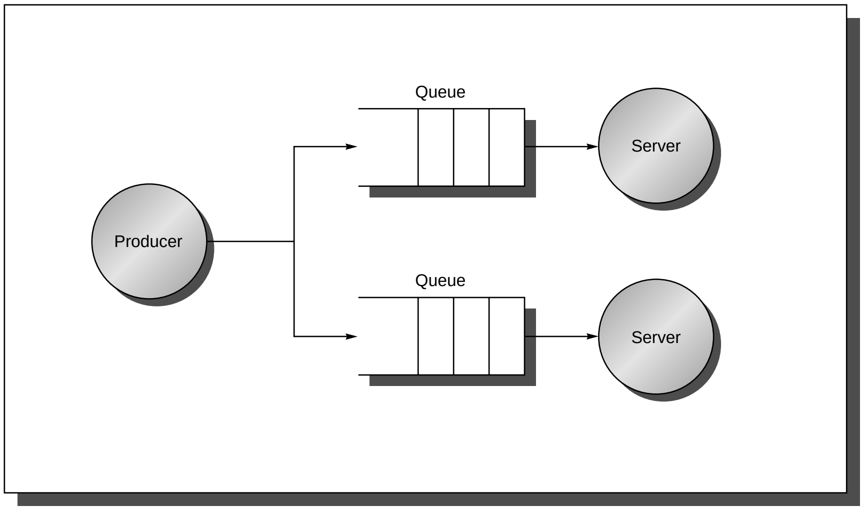


FIGURE 6.18 The single-producer, single-server model of Figure 6.16 is extended with another server and buffer. This increases I/O system throughput and takes less time to service producer tasks. Increasing the number of servers is a common technique in I/O systems. There is a potential imbalance problem with two buffers: Unless data is placed perfectly in the buffers, sometimes one server will be idle with an empty buffer while the other server is busy with many tasks in its buffer.

1. *Entry time*—The time for the user to enter the command. The graphics system in Figure 6.19 took 0.25 seconds on average to enter a command versus 4.0 seconds for the keyboard system.
2. *System response time*—The time between when the user enters the command and the complete response is displayed.
3. *Think time*—The time from the reception of the response until the user begins to enter the next command.

The sum of these three parts is called the *transaction time*. Several studies report that user productivity is inversely proportional to transaction time; *transactions per hour* is a measure of the work completed per hour by the user.

The results in Figure 6.19 show that reduction in response time actually decreases transaction time by more than just the response time reduction: Cutting system response time by 0.7 seconds saves 4.9 seconds (34%) from the conventional transaction and 2.0 seconds (70%) from the graphics transaction. This implausible result is explained by human nature: People need less time to think when given a faster response.

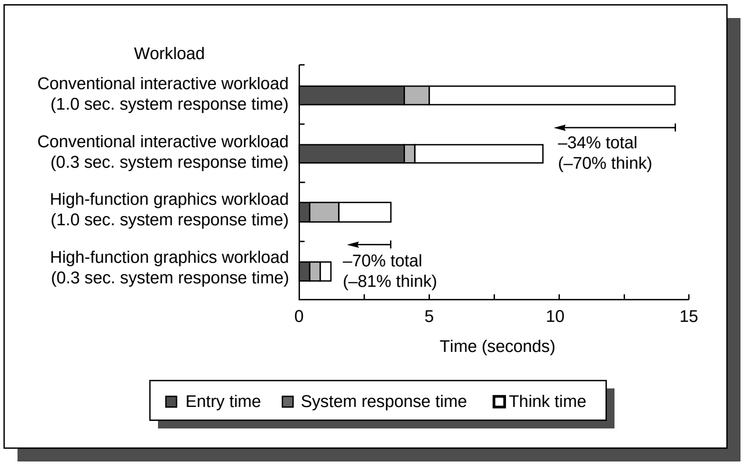


FIGURE 6.19 A user transaction with an interactive computer divided into entry time, system response time, and user think time for a conventional system and graphics system. The entry times are the same, independent of system response time. The entry time was 4 seconds for the conventional system and 0.25 seconds for the graphics system. (From Brady [1986].)

Whether these results are explained as a better match to the human attention span or getting people "on a roll," several studies report this behavior. In fact, as computer responses drop below one second, productivity seems to make a more than linear jump. Figure 6.20 compares transactions per hour (the inverse of transaction time) of a novice, an average engineer, and an expert performing physical design tasks on graphics displays. System response time magnified talent: a novice with subsecond response time was as productive as an experienced professional with slower response, and the experienced engineer in turn could outperform the expert with a similar advantage in response time. In all cases the number of transactions per hour jumps more than linearly with subsecond response time.

Since humans may be able to get much more work done per day with better response time, it is possible to attach an economic benefit to the customer of lowering response time into the subsecond range [IBM 1982], thereby helping the architect decide how to tip the balance between response time and throughput.

A Little Queuing Theory

With an appreciation of the importance of response time, we can give a set of simple theorems that will help calculate response time and throughput of an entire I/O system. Let's start with a black box approach to I/O systems, as in Figure 6.21. In our example the CPU is making I/O requests that arrive at the I/O device, and the requests "depart" when the I/O device fulfills them.

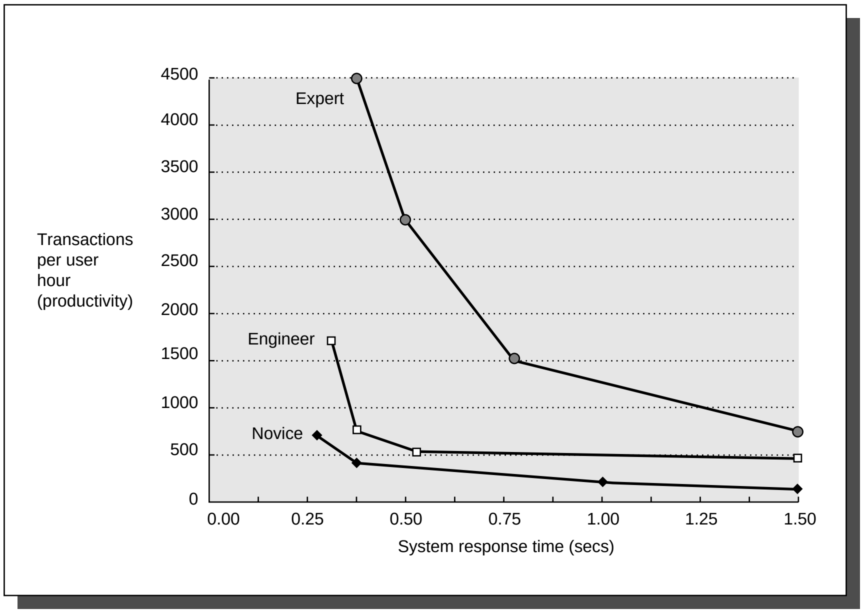


FIGURE 6.20 Transactions per hour versus computer response time for a novice, experienced engineer, and expert doing physical design on a graphics system. *Transactions per hour* is a measure of productivity. (From IBM [1982].)

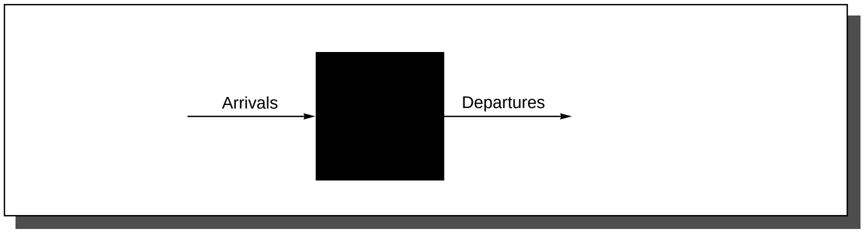


FIGURE 6.21 Treating the I/O system as a black box. This leads to a simple but important observation: If the system is in steady state, then the number of tasks entering the systems must equal the number of tasks leaving the system.

We are usually interested in the long term, or steady state, of a system rather than in the initial start-up conditions. Hence we make the simplifying assumption that we are evaluating systems in equilibrium: the input rate must be equal to the output rate. This leads us to *Little's Law*, which relates the average number of tasks in the system, the average arrival rate of new tasks, and the average time to perform a task:

$$\text{Mean number of tasks in system} = \text{Arrival rate} \times \text{Mean response time}$$

Little’s Law applies to any system in equilibrium, as long as nothing inside the black box is creating new tasks or destroying them. This simple equation is surprisingly powerful, as we shall see.

If we open the black box, we see Figure 6.22. The areas where the tasks accumulate, waiting to be serviced, is called the *queue,* or *waiting line,* and the device performing the requested service is called the *server.*

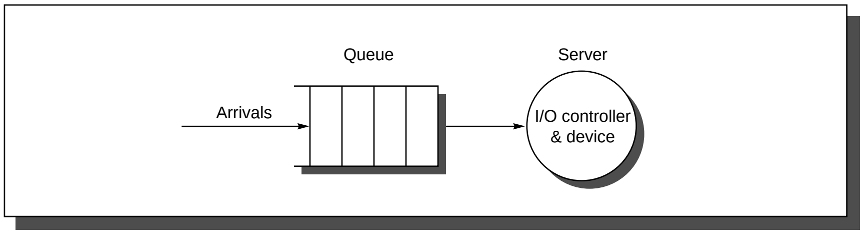


FIGURE 6.22 The single server model for this section. In this situation an I/O request “departs” by being completed by the server.

Little’s Law and a series of definitions lead to several useful equations:

$\text{Time}_{\text{server}}$—Average time to service a task; service rate is $1/\text{Time}_{\text{server}}$, traditionally represented by the symbol μ in many texts.

$\text{Time}_{\text{queue}}$—Average time per task in the queue.

$\text{Time}_{\text{system}}$—Average time/task in the system, or the response time, the sum of $\text{Time}_{\text{queue}}$ and $\text{Time}_{\text{server}}$.

Arrival rate—Average number of arriving tasks/second, traditionally represented by the symbol λ in many texts.

$\text{Length}_{\text{server}}$—Average number of tasks in service.

$\text{Length}_{\text{queue}}$—Average length of queue.

$\text{Length}_{\text{system}}$—Average number of tasks in system, the sum of $\text{Length}_{\text{queue}}$ and $\text{Length}_{\text{server}}$.

One common misunderstanding can be made clearer by these definitions: whether the question is how long a task must wait in the queue before service starts ($\text{Time}_{\text{queue}}$) or how long a task takes until it is completed ($\text{Time}_{\text{system}}$). The latter term is what we mean by response time, and the relationship between the terms is $\text{Time}_{\text{system}} = \text{Time}_{\text{queue}} + \text{Time}_{\text{server}}$.

Using the terms above we can restate Little’s Law as

$$\text{Length}_{\text{system}} = \text{Arrival rate} \times \text{Time}_{\text{system}}$$

We can also talk about how busy a system is from these definitions. Server utilization is simply

$$\text{Server utilization} = \frac{\text{Arrival rate}}{\text{Service rate}}$$

The value must be between 0 and 1, for otherwise there would be more tasks arriving than could be serviced, violating our assumption that the system is in equilibrium. Utilization is also called *traffic intensity* and is represented by the symbol ρ in many texts.

EXAMPLE Suppose an I/O system with a single disk gets about 10 I/O requests per second and the average time for a disk to service an I/O request is 50 ms. What is the utilization of the I/O system?

ANSWER The service rate is then

$$\frac{1}{50 \text{ ms}} = \frac{1}{0.05 \text{ sec}} = 20 \text{ I/O per second (IOPS)}$$

Using the equation above,

$$\text{Server utilization} = \frac{\text{Arrival rate}}{\text{Service rate}} = \frac{10 \text{ IOPS}}{20 \text{ IOPS}} = 0.50$$

So the I/O system utilization is 0.5. ■

Little's Law can be applied to the components of the black box as well, since they must also be in equilibrium:

$$\text{Length}_{\text{queue}} = \text{Arrival rate} \times \text{Time}_{\text{queue}}$$
$$\text{Length}_{\text{server}} = \text{Arrival rate} \times \text{Time}_{\text{server}}$$

EXAMPLE Suppose the average time to satisfy a disk request is 50 ms and the I/O system with many disks gets about 200 I/O requests per second. What is the mean number of I/O requests at the disk server?

ANSWER Using the equation above,

$$\text{Length}_{\text{server}} = \text{Arrival rate} \times \text{Time}_{\text{server}} = \frac{200}{\text{sec}} \times 0.05 \text{ sec} = 10$$

So there are 10 requests on average at the disk server. ■

How the queue delivers tasks to the server is called the *queue discipline*. The simplest and most common discipline is *first-in-first-out* (FIFO). If we assume FIFO we can relate time waiting in the queue to the mean number of tasks in the queue:

$$\text{Time}_{\text{system}} = \text{Length}_{\text{queue}} \times \text{Time}_{\text{server}} + \text{Mean time to complete service of tasks when new task arrives}$$

That is, the system response time is the number of tasks in the queue times the mean service time plus the time it takes the server to complete whatever tasks are being serviced when a new task arrives.

The last component of the equation is not as simple as it first appears. A new task can arrive at any instant, so we have no basis to know how long the existing task has been in the server. Although such requests are random events, if we know something about the distribution of events we can predict performance.

To estimate this answer we need to know a little about distributions of *random variables*. A variable is random if it takes one of a specified set of values with a specified probability; that is, you cannot know exactly what its next value will be, but you do know the probability of all possible values.

One way to characterize the distribution of values of a random variable is a *histogram*, which divides the range between the minimum and maximum values into subranges called *buckets*. Histograms then plot the number in each bucket as columns. Histograms work well for distributions that are discrete values—for example, the number of I/O requests. For distributions that are not discrete values, such as time waiting for an I/O request, we need a curve to plot the values over the full range so that we can accurately estimate the value. Stated alternatively, we need a histogram with an infinite number of buckets.

Hence, to be able to solve the last part of the equation above we need to characterize the distribution of this random variable. The mean time and some measure of the variance is sufficient for that characterization. For the first term we use the *weighted arithmetic mean time* (see page 26 in Chapter 1 for a slightly different version of the formula):

$$\text{Weighted mean time} = \frac{f_1 \times T_1 + f_2 \times T_2 + \ldots + f_n \times T_n}{f_1 + f_2 + \ldots + f_n}$$

where T_i is the time for task *i* and f_i is the frequency of occurrence of task *i*.

To characterize variability about the mean, many people use the standard deviation. Let's use the *variance* instead, which is simply the square of the standard deviation. Given the weighted mean, the variance can be calculated as

$$\text{Variance} = \frac{f_1 \times T_1^2 + f_2 \times T_2^2 + \ldots + f_n \times T_n^2}{f_1 + f_2 + \ldots + f_n} - \text{Weighted mean time}^2$$

The problem with variance is that you must remember the units. Let's assume the distribution is of time. If time is on the order of 100 milliseconds, then squaring it yields 10,000 square milliseconds. This unit is certainly unusual. It would be more convenient if we had a unitless measure.

To avoid this unit problem, we use the *squared coefficient of variance*, traditionally called *C:*

$$C = \frac{\text{Variance}}{\text{Weighted mean time}^2}$$

For reasons stated earlier, we are trying to characterize random events, but to be able to predict performance we need random events with certain nice properties. Figure 6.23 gives a few examples. An *exponential distribution*, with most of the times short relative to average but with a few long ones, has a C value of 1. In a *hypoexponential distribution,* most values are close to average and C is less than 1. In a *hyperexponential distribution,* most values are further from the average and C is greater than 1. The disk service is best measured with a C of about 1.5. As we shall see, the value of C affects the simplicity of the queuing formulas.

Distribution type	**C**	**% less than average**	**90% of distribution is less than**
Hypoexponential	0.5	57%	2.0 times average
Exponential	1.0	63%	2.3 times average
Hyperexponential	2.0	69%	2.8 times average

FIGURE 6.23 Examples of value of squared coefficient of variance C and variability of distributions given an unlimited number of tasks (*infinite population*).

Note that we are using a constant to characterize variability about the mean. Since C does not vary over time, the past history of events has no impact on the probability of an event occurring now. This forgetful property is called *memoryless* and is a key assumption used to predict behavior.

Finally, we can answer the question about the length of time a new task must wait for the server to complete a task, called the *average residual service time*:

$$\text{Average residual service time} = 1/2 \times \text{Weighted mean time} \times (1 + C)$$

Although we won't derive this formula, we can appeal to intuition. When the distribution is not random and all possible values are equal to the average, the variance is 0 and so C is 0. The average residual service time is then just half the average service time, as we would expect.

EXAMPLE Using the definitions and formulas above, derive the average time waiting in the queue ($\text{Time}_{\text{queue}}$) in terms of the average service time ($\text{Time}_{\text{server}}$), server utilization, and the squared coefficient of variance (C).

ANSWER All tasks in the queue ($\text{Length}_{\text{queue}}$) ahead of the new task must be completed before the task can be serviced; each takes on average $\text{Time}_{\text{server}}$. If a task is at the server, it takes average residual service time to complete. The chance the server is busy is *server utilization*, hence the expected time for service is Server utilization $\times$ Average residual service time. This leads to our initial formula:

$$\text{Time}_{\text{queue}} = \text{Length}_{\text{queue}} \times \text{Time}_{\text{server}} + \text{Server utilization} \times \text{Average residual service time}$$

Replacing average residual service time by its definition and $\text{Length}_{\text{queue}}$ by Arrival rate $\times$ $\text{Time}_{\text{queue}}$ yields

$$\text{Time}_{\text{queue}} = \text{Server utilization} \times (1/2 \times \text{Time}_{\text{server}} \times (1 + C)) + (\text{Arrival rate} \times \text{Time}_{\text{queue}}) \times \text{Time}_{\text{server}}$$

Rearranging the last term, let us replace Arrival rate $\times$ $\text{Time}_{\text{server}}$ by Server utilization since

$$\text{Server utilization} = \frac{\text{Arrival rate}}{1/\text{Time}_{\text{server}}} = \text{Arrival rate} \times \text{Time}_{\text{server}}$$

It works as follows:

$$\begin{aligned}\text{Time}_{\text{queue}} &= \text{Server utilization} \times (1/2 \times \text{Time}_{\text{server}} \times (1 + C)) + (\text{Arrival rate} \times \text{Time}_{\text{server}}) \times \text{Time}_{\text{queue}} \\ &= \text{Server utilization} \times (1/2 \times \text{Time}_{\text{server}} \times (1 + C)) + \text{Server utilization} \times \text{Time}_{\text{queue}}\end{aligned}$$

Rearranging terms and simplifying gives us the desired equation:

$$\text{Time}_{\text{queue}} = \text{Server utilization} \times (1/2 \times \text{Time}_{\text{server}} \times (1 + C)) + \text{Server utilization} \times \text{Time}_{\text{queue}}$$

$$\text{Time}_{\text{queue}} - \text{Server utilization} \times \text{Time}_{\text{queue}} = \text{Server utilization} \times (1/2 \times \text{Time}_{\text{server}} \times (1 + C))$$

$$\text{Time}_{\text{queue}} \times (1 - \text{Server utilization}) = \frac{\text{Server utilization} \times (\text{Time}_{\text{server}} \times (1 + C))}{2}$$

$$\text{Time}_{\text{queue}} = \frac{\text{Time}_{\text{server}} \times (1 + C) \times \text{Server utilization}}{2 \times (1 - \text{Server utilization})}$$

Note that when we have an exponential distribution, then C = 1.0, so this formula simplifies to

$$\text{Time}_{\text{queue}} = \text{Time}_{\text{server}} \times \frac{\text{Server utilization}}{(1 - \text{Server utilization})}$$

■

These equations and this subsection are based on an area of applied mathematics called *queuing theory,* which offers equations to predict behavior of such random variables. Real systems are too complex for queuing theory to provide exact analysis, and hence queuing theory works best when only approximate answers are needed. This subsection is a simple introduction, and interested readers can find many books on the topic.

Requests for service from an I/O system can be modeled by a random variable, because the operating system is normally switching between several processes that generate independent I/O requests. We also model I/O service times by a random variable given the probabilistic nature of disks in terms of seek and rotational delays.

Queuing theory makes a sharp distinction between past events, which can be characterized by measurements using simple arithmetic, and future events, which are predictions requiring mathematics. In computer systems we commonly predict the future from the past; one example is least recently used block placement (see Chapter 5). Hence the distinction between measurements and predicted distributions is often blurred here, and we use measurements to verify the type of distribution and then rely on the distribution thereafter.

Let’s review the assumptions about the queuing model:

- The system is in equilibrium.
- The times between two successive requests arriving, called the *interarrival times*, are exponentially distributed.
- The number of requests is unlimited (this is called an *infinite population model* in queuing theory).
- The server can start on the next customer immediately after finishing with the prior one.
- There is no limit to the length of the queue, and it follows the first-in-first-out order discipline.
- All tasks in line must be completed.

Such a queue is called *M/G/1*:

M = exponentially random request arrival (C = 1), with M standing for the memoryless property mentioned above

G = general service distribution (i.e., not necessarily exponential)

1 = single server

When service times are exponentially distributed, this model becomes an *M/M/1* queue and we can use the simple equation for waiting time at the end of the last example. The M/M/1 model is a simple and widely used model.

The assumption of exponential distribution is commonly used in queuing examples for two reasons, one good and one bad. The good reason is that a collection of many arbitrary distributions acts as an exponential distribution. Many times in computer systems a particular behavior is the result of many components interacting, so an exponential distribution of interarrival times is the right model. The bad reason is that the math is simpler if you assume exponential distributions.

Let's put queuing theory to work in a few Examples.

EXAMPLE Suppose a processor sends 10 disk I/Os per second, these requests are exponentially distributed, and the average disk service time is 20 ms. Answer the following questions:

1. On average, how utilized is the disk?
2. What is the average time spent in the queue?
3. What is the 90th percentile of the queuing time?
4. What is the average response time for a disk request, including the queuing time and disk service time?

ANSWER Let's restate these facts:

Average number of arriving tasks/second is 10.

Average disk time to service a task is 20 ms (0.02 sec).

The server utilization is then

$$\text{Server utilization} = \frac{\text{Arrival rate}}{\text{Service rate}} = \frac{10}{1/0.02} = 0.2$$

Since the interarrival times are exponentially distributed, we can use the simplified formula for the average time spent waiting in line:

$$\text{Time}_{\text{queue}} = \text{Time}_{\text{server}} \times \frac{\text{Server utilization}}{(1 - \text{Server utilization})} = 20\text{ ms} \times \frac{0.2}{1 - 0.2} = 20 \times \frac{0.2}{0.8} = 20 \times 0.25 = 5\text{ ms}$$

From Figure 6.23 (page 512), the 90th percentile is 2.3 times the mean waiting time, so it is 11.5 ms. The average response time is

$$\text{Time}_{\text{queue}} + \text{Time}_{\text{server}} = 5 + 20\text{ ms} = 25\text{ ms}$$

■

EXAMPLE Suppose we get a new, faster disk. Recalculate the answers to the questions above, assuming the disk service time is 10 ms.

ANSWER The disk utilization is then

$$\text{Server utilization} = \frac{\text{Arrival rate}}{\text{Service rate}} = \frac{10}{1/0.01} = 0.1$$

Since the service distribution is exponential, we can use the simplified formula for the average time spent waiting in line:

$$\text{Time}_{\text{queue}} = \text{Time}_{\text{server}} \times \frac{\text{Server utilization}}{(1 - \text{Server utilization})} = 10\text{ ms} \times \frac{0.1}{1-0.1} = 10 \times \frac{0.1}{0.9} = 10 \times 0.11 = 1.11\text{ ms}$$

The 90th percentile of the mean waiting time is 2.56 ms.

The average response time is 10 + 1.11 ms or 11.11 ms, 2.25 times faster than the old response time even though the new service time is only 2.0 times faster. ■

Section 6.7 has more examples using queuing theory to predict performance.

Examples of Benchmarks of Disk Performance

The prior subsection tries to predict the performance of storage subsystems. We also need to measure the performance of real systems to collect the values of parameters needed for prediction, to determine if the queuing theory assumptions hold, and to suggest what to do if the assumptions don't hold.

This subsection describes three benchmarks, each illustrating novel concerns regarding storage systems versus processors.

Transaction Processing Benchmarks

Transaction processing (TP, or OLTP for on-line transaction processing) is chiefly concerned with *I/O rate*: the number of disk accesses per second, as opposed to *data rate*, measured as bytes of data per second. TP generally involves changes to a large body of shared information from many terminals, with the TP system guaranteeing proper behavior on a failure. If, for example, a bank's computer fails when a customer withdraws money, the TP system would guarantee that the account is debited if the customer received the money and that the account is unchanged if the money was not received. Airline reservations systems as well as banks are traditional customers for TP.

Two dozen members of the TP community conspired to form a benchmark for the industry and, to avoid the wrath of their legal departments, published the report anonymously [1985]. This benchmark, called *DebitCredit*, simulates bank tellers and has as its bottom line the number of debit/credit transactions per second (TPS). The DebitCredit performs the operation of a customer depositing or withdrawing money. *TPC-A* and *TPC-B* are more tightly specified versions of this original benchmark. The organization responsible for standardizing TPC-A and TPC-B have also developed benchmarks on complex query processing (*TPC-C*) and decision support (*TPC-D*).

Disk I/O for DebitCredit is random reads and writes of 100-byte records along with occasional sequential writes. Depending on how cleverly the transaction-processing system is designed, each transaction results in between 2 and 10 disk I/Os and takes between 5000 and 20,000 CPU instructions per disk I/O. The variation depends largely on the efficiency of the transaction-processing software, although in part it depends on the extent to which disk accesses can be avoided by keeping information in main memory. Hence, TPC measures the database software as well as the underlying machine.

The main performance measurement is the peak TPS, under the restriction that 90% of the transactions have less than a two-second response time. The benchmark requires that for TPS to increase, the number of tellers and the size of the account file must also increase. Figure 6.24 shows this unusual relationship in which more TPS requires more users. This scaling is to ensure that the benchmark really measures disk I/O; otherwise a large main memory dedicated to a database cache with a small number of accounts would unfairly yield a very high TPS. (Another perspective is that the number of accounts must grow, since a person is not likely to use the bank more frequently just because the bank has a faster computer!)

TPS	Number of ATMs	Account file size
10	1000	0.1 GB
100	10,000	1.0 GB
1000	100,000	10.0 GB
10,000	1,000,000	100.0 GB

FIGURE 6.24 Relationship among TPS, tellers, and account file size. The DebitCredit benchmark requires that the computer system handle more tellers and larger account files before it can claim a higher transaction-per-second milestone. The benchmark is supposed to include "terminal handling" overhead, but this metric is sometimes ignored.

Another novel feature of TPC-A and TPC-B is that they address how to compare the performance of systems with different configurations. In addition to reporting TPS, benchmarkers must also report the cost per TPS, based on the five-year cost of the computer system hardware and software.

SPEC System-Level File Server (SFS) Benchmark

The SPEC benchmarking effort is best known for its characterization of processor performance, but it branches out into other fields as well. In 1990 seven companies agreed on a synthetic benchmark, called SFS, to evaluate systems running the Sun Microsystems network file service NFS. This synthetic mix was based on measurements on NFS systems to propose a reasonable mix of reads, writes, and file operations such as examining a file. SFS supplies default parameters for comparative performance: For example, half of all writes are done in 8-KB blocks and half are done in partial blocks of 1, 2, or 4 KB. For reads the mix is 85% full blocks and 15% partial blocks.

Like TPC-B, SFS scales the size of the file system according to the reported throughput: For every 100 NFS operations per second, the capacity must increase by 1 GB. It also limits the average response time, in this case to 50 ms. Figure 6.25 shows average response time versus throughput for three systems. Unfortunately, unlike TPC-B, SFS does not normalize for different configurations. The fastest system in Figure 6.25 has 12 times the number of CPUs and disks as the slowest system, but SPEC leaves it to you to calculate price versus performance.

Self-Scaling I/O Benchmark

A different approach to I/O performance analysis was proposed by Chen and Patterson [1994b]. The first step is a *self-scaling benchmark*, which automatically and dynamically adjusts *several* aspects of its workload according to the performance characteristics of the system being measured. By doing so, the benchmark automatically scales across current and future systems. This scaling is more general than the scaling found in TPC-B and SFS, for scaling here varies five parameters, according to the characteristics of the system being measured, rather than just one.

This first step aids in understanding system performance by reporting how performance varies according to each of five workload parameters. These five parameters determine the first-order performance effects in I/O systems:

1. *Number of unique bytes touched*—This is the number of unique data bytes read or written in a workload; essentially, it is the total size of the data set.
2. *Percentage of reads.*
3. *Average I/O request size*—It chooses sizes from a distribution with a coefficient of variance (C) of one.
4. *Percentage of sequential requests*—This is the percentage of requests that sequentially follow the prior request. When set at 50%, on average half of the accesses are to the next sequential address.
5. *Number of processes*—This is the concurrency in the workload, that is, the number of processes simultaneously issuing I/O.

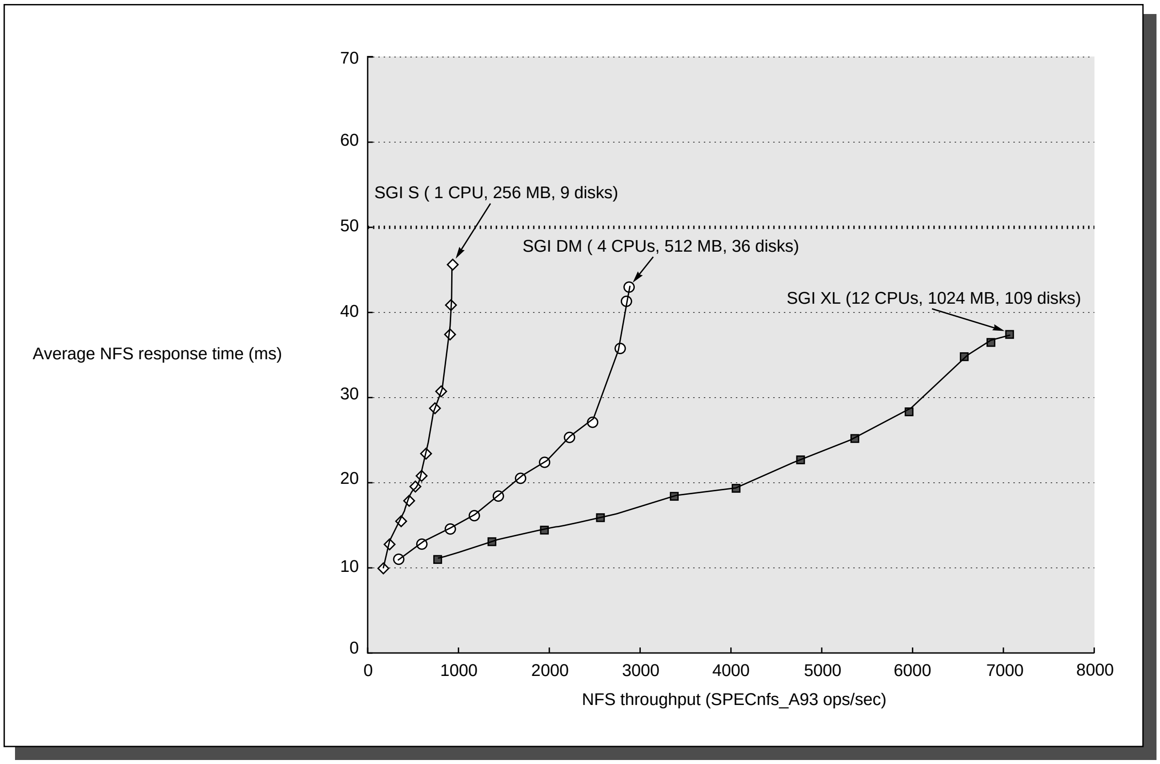


FIGURE 6.25 SPEC SFS performance for three SGI Challenge servers. The dashed line represents the 50-ms average response time limit imposed by SPEC. Reported in March 1995, these systems all ran IRIX version 5.3 with the EFS file system and all used the R4400 microprocessor. The XL model processors ran at 200 MHz and the other two used 150 MHz. Each system had one 1-GB disk with the rest being 2-GB disks, most spinning at 7200 RPM. SPEC SFS also divides the peak rate by 10 and calls this quotient SPECnsf_A93 users/second. The numbers of such users per second for these three machines are 84, 283, and 702, respectively.

The benchmark first chooses a nominal value for each of the five parameters based on the system's performance. It then varies each parameter in turn while the other four parameters remain at their fixed, nominal values. The one exception is the first parameter, since it determines whether all accesses go to the file cache or to disk. Because of the very different performance for file cache and disk accesses, the benchmark automatically picks two values for the number of bytes accessed.

The resulting I/O performance is then plotted for each of the parameters. Figure 6.26 shows the performance for workstations and mainframes, using the nominal parameter values collected by the self-scaling benchmark as a function of unique bytes touched. These plots give insight into appropriate workloads and resulting performance. The width of the high-performance parts of the curves is determined by the size of the file cache. For example, the HP 730 offers the highest performance, provided the workload fits in its small file cache, and workloads that would need to go to disk on other systems can be satisfied by the very large file cache of the Convex.

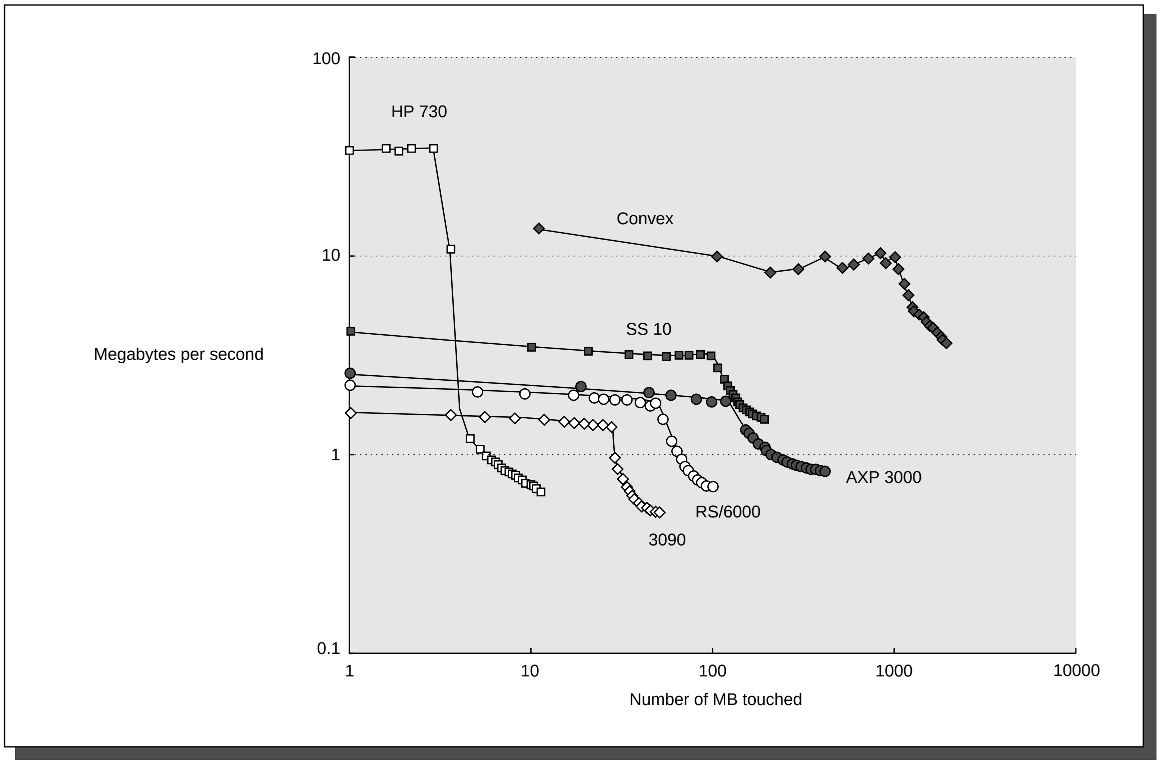


FIGURE 6.26 Performance versus megabytes touched for several workstations and mainframes (see section 6.8). Note the log-log scale. These results use the nominal values selected by the self-scaling benchmark. For example, 50% of accesses are reads and 50% are writes. The primary difference between the systems is the average access size of 120 KB for the Convex; adjusting for a common access size would halve Convex performance but make little change to the other lines in this plot.

The self-scaling benchmark increases our understanding of a system and scales the workload to remain relevant as technology advances. It complicates the task of comparing results from two systems, however. The problem is that the benchmark may choose different workloads on which to measure each system.

Hence, the second part of this new approach is to estimate the performance of other workloads. It estimates performance for unmeasured workloads by assuming that the *shape* of a performance curve for one parameter is independent of the values of the other parameters. This assumption leads to an overall performance equation of

$$\mathrm{Perf}(X, Y, Z, \ldots) = \mathrm{Perf}(X_{\mathrm{nominal}}, Y_{\mathrm{nominal}}, Z_{\mathrm{nominal}}, \ldots) \times f_X(X) \times f_Y(Y) \times f_Z(Z) \times \ldots$$

where X, Y, Z, ... are the parameters. Suppose the nominal values were 50% reads and 50% of accesses as sequential, but the desired workload had 60% reads and 60% sequential, and all other parameters matched the nominal values. The predicted performance is the nominal performance multiplied by the measured ratio of 60% reads to 50% reads and by the measured ratio at 60% sequential to 50% sequential. Chen and Patterson [1994b] have shown that this technique yields accurate performance estimates, within 10% for most workloads.

We use this benchmark to evaluate systems in section 6.8.

6.5 Reliability, Availability, and RAID

Although throughput and response time have their analogues in processor design, reliability is given considerably more attention in storage than in processors. This brings us to two terms that are often confused—*reliability* and *availability.* The term reliability is commonly used interchangeably with availability: if something breaks, but the user can still use the system, it seems as if the system still works and hence it seems more reliable. Here is a clearer distinction:

Reliability—Is anything broken?

Availability—Is the system still available to the user?

Adding hardware can therefore improve availability (for example, ECC on memory), but it cannot improve reliability (the DRAM is still broken). Reliability can only be improved by bettering environmental conditions, by building from more reliable components, or by building with fewer components. Another term, *data integrity*, refers to consistent reporting when information is lost because of failure; this is very important to some applications.

One innovation that improves both availability and performance of storage systems is *disk arrays*. The argument for arrays is that since price per megabyte is independent of disk size, potential throughput can be increased by having many disk drives and, hence, many disk arms. For example, Figure 6.25 (page 519) shows how NFS throughput increases as the systems expand from 9 disks to 109 disks. Simply spreading data over multiple disks, called *striping*, automatically forces accesses to several disks. (Although arrays improve throughput, latency is not necessarily improved.) The drawback to arrays is that with more devices, reliability drops: *N* devices generally have 1/*N* the reliability of a single device.

So, while a disk array can never be more reliable than a smaller number of larger disks when each disk has the same failure rate, availability can be improved by adding redundant disks. That is, if a single disk fails, the lost information can be reconstructed from redundant information. The only danger is in having another disk failure between the time a disk fails and the time it is replaced (termed *mean time to repair*, or MTTR). Since the *mean time to failure* (MTTF) of disks is five or more years, and the MTTR is measured in hours, redundancy can make the availability of 100 disks much higher than that of a single disk. These systems have become known by the acronym *RAID*, standing for *redundant array of inexpensive disks*.

There are several approaches to redundancy that have different overhead and performance. Figure 6.27 shows the RAID levels and gives an example of how eight disks would have to be supplemented by redundant or check disks at each level plus the number of failures that the system would survive.

RAID level		**Failures survived**	**Data disks**	**Check disks**
0	Nonredundant	0	8	0
1	Mirrored	1	8	8
2	Memory-style ECC	1	8	4
3	Bit-interleaved parity	1	8	1
4	Block-interleaved parity	1	8	1
5	Block-interleaved distributed parity	1	8	1
6	P+Q redundancy	2	8	2

FIGURE 6.27 RAID levels, their availability, and their overhead in redundant disks. The paper that introduced the term RAID [Patterson, Gibson, and Katz 1987] used a numerical classification for these schemes that has become popular; in fact, the nonredundant disk array is sometimes called RAID 0.

One problem is discovering when a disk fails. Fortunately, magnetic disks provide information about their correct operation. There is extra information recorded in each sector to discover errors within that sector. As long as we transfer at least one sector and check the error detection information when reading sectors, electronics associated with disks will with very high probability discover when a disk fails or loses information.

We cover here the most popular of these RAID levels; readers interested in more detail should see the paper by Chen et al. [1994].

Mirroring (RAID 1)

The traditional solution to disk failure, called *mirroring* or *shadowing*, uses twice as many disks. Whenever data is written to one disk, that data is also written to a redundant disk, so that there are always two copies of the information. If a disk fails, the system just goes to the "mirror" to get the desired information. Mirroring is the most expensive solution.

Bit-Interleaved Parity (RAID 3)

The cost of higher availability can be reduced to $1/N$, where N is the number of disks in a protection group. Rather than have a complete copy of the original data for each disk, we need only add enough redundant information to restore the lost information on a failure. Reads or writes go to all disks in the group, with one extra disk to hold the check information in case there is a failure.

Parity is one such scheme. Readers unfamiliar with parity can think of the redundant disk as having the sum of all the data in the other disks. When a disk fails, then you subtract all the data in the good disks from the parity disk; the remaining information must be the missing information. Parity is simply the sum modulo 2. The assumption is that failures are so rare that taking longer to recover from failure but reducing redundant storage is a good trade-off.

Just as direct-mapped associative placement in caches can be considered a special case of set-associative placement (see section 5.2), the mirroring can be considered the special case of one data disk and one parity disk ($N = 1$). Parity can be accomplished by duplicating the data, so mirrored disks have the advantage of simplifying parity calculation. Duplicating data also means that the controller can improve read performance by reading from the disk of the pair that has the shortest seek distance, although this optimization means writes must wait for the arm with the longer seek since arms are no longer synchronized. Of course, the redundancy of $N = 1$ has the highest overhead for increasing disk availability.

Block-Interleaved Distributed Parity (RAID 5)

This level uses the same organization of disks, but data is accessed differently. In the prior organization every access went to all disks. Some applications would prefer to do smaller accesses, allowing independent accesses to occur in parallel. That is the purpose of this next RAID level. Since error-detection information in each sector is checked on reads to see if data is correct, such "small reads" to each disk can occur independently as long as the minimum access is one sector.

Writes are another matter. It would seem that each small write would demand that all other disks be accessed to read the rest of the information needed to recalculate the new parity. In our example, a "small write" would require reading the other three data disks, adding the new information, and then writing the new parity to the parity disk and the new data to the data disk. The key insight to reduce this overhead is that parity is simply a sum of information; by watching which bits change when we write the new information, we need only change the corresponding bits on the parity disk. We must read the old data, compare old data to the new data to see which bits change, read the old parity, change the corresponding bits, then write the new data and new parity. Thus the small write involves four disk accesses for two disks instead of accessing all disks.

This scheme supports mixtures of large reads, large writes, small reads, and small writes. One drawback to the system is that the parity disk must be updated on every write, so it is the bottleneck for sequential writes. To fix the parity-write bottleneck, the parity information is spread throughout all the disks so that there is no single bottleneck for writes. Figure 6.28 shows how data are distributed in this disk array organization.

As the organization on the right shows, the parity associated with each row of data blocks is no longer restricted to a single disk. This organization allows for multiple writes to occur simultaneously as long as the stripe units are not located in the same disks. For example, a write to block 8 on the right must also access its parity block P2, thereby occupying the first and third disks. A second write to block 5 on the right, implying an update to its parity block P1, accesses the second and fourth disks and thus could occur at the same time as the prior write.

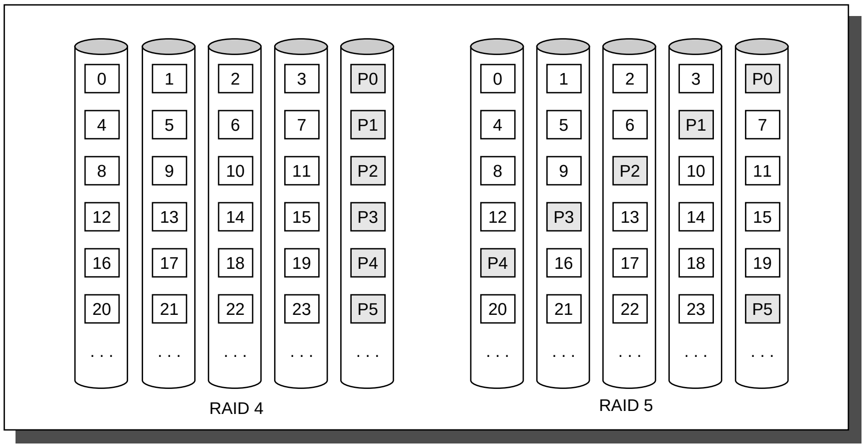


FIGURE 6.28 Block-interleaved parity (RAID 4) versus distributed block-interleaved parity (RAID 5). By distributing parity blocks to all disks, some small writes can be performed in parallel.

The higher throughput, measured either as megabytes per second or as I/Os per second, and the ability to recover from failures make RAID attractive. When combined with the advantages of smaller volume and lower power of small-diameter drives, RAIDs are playing an increasing role in storage systems.

6.6 Crosscutting Issues: Interfacing to an Operating System

In a manner analogous to the way compilers use an instruction set, operating systems control what I/O techniques implemented by the hardware will actually be used. For example, many I/O controllers used in early UNIX systems were 16-bit microprocessors. To avoid problems with 16-bit addresses in controllers, UNIX was changed to limit the maximum I/O transfer to 63 KB or less. Thus, a new I/O controller designed to efficiently transfer 1-MB files would never see more than 63 KB at a time under UNIX, no matter how large the files.

Caches Cause Problems for Operating Systems—Stale Data

The prevalence of caches in computer systems has added to the responsibilities of the operating system. Caches imply the possibility of two copies of the data—one each for cache and main memory—while virtual memory can result in three copies—for cache, memory, and disk. This brings up the possibility of *stale data*: the CPU or I/O system could modify one copy without updating the other copies (see section 5.9). Either the operating system or the hardware must make sure that the CPU reads the most recently input data and that I/O outputs the correct data, in the presence of caches and virtual memory. Whether the stale-data problem arises depends in part on where the I/O is connected to the computer. If it is connected to the CPU cache, as shown in Figure 6.29, there is no stale-data problem; all I/O devices and the CPU see the most accurate version in the cache, and existing mechanisms in the memory hierarchy ensure that other copies of the data will be updated. The side effect is lost CPU performance, since I/O will replace blocks in the cache with data that are unlikely to be needed by the process running in the CPU at the time of the transfer. In other words, all I/O data goes through the cache, but little of it is referenced. This arrangement also requires arbitration between the CPU and I/O to decide who accesses the cache. If I/O is connected to memory, as in Figure 6.15 (page 502), then it doesn't interfere with CPU, provided the CPU has a cache. In this situation, however, the stale-data problem occurs. Alternatively, I/O can just invalidate data—either all data that might match (no tag check) or only data that matches.

There are two parts to the stale-data problem:

1. The I/O system sees stale data on output because memory is not up-to-date.
2. The CPU sees stale data in the cache on input after the I/O system has updated memory.

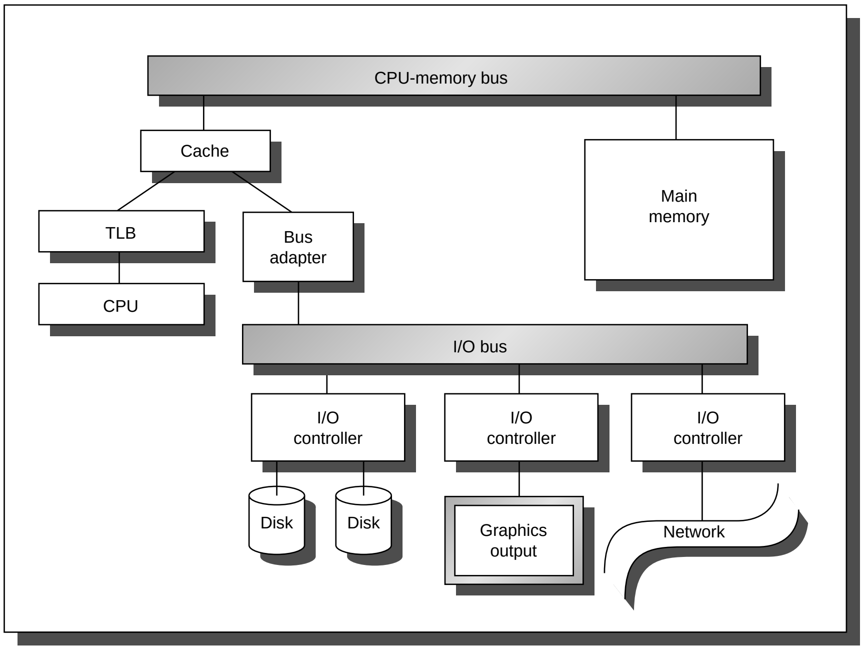


FIGURE 6.29 Example of I/O connected directly to the cache.

The first dilemma is how to output correct data if there is a cache and I/O is connected to memory. A write-through cache solves this by ensuring that memory will have the same data as the cache. A write-back cache requires the operating system to flush output addresses to make sure they are not in the cache. This flush takes time, even if the data is not in the cache, since address checks are sequential. Alternatively, the hardware can check cache tags during output to see if they are in a write-back cache, and only interact with the cache if the output tries to read data that is in the cache.

The second problem is ensuring that the cache won't have stale data after input. The operating system can guarantee that the input data area can't possibly be in the cache. If it can't guarantee this, the operating system flushes input addresses to make sure they are not in the cache. Again, this takes time, whether or not the input addresses are in the cache. As before, extra hardware can be added to check tags during an input and invalidate the data if there is a conflict. These problems are basically the same as cache coherency in a multiprocessor, discussed in Chapter 8; I/O can be thought of as a second dedicated processor in a multiprocessor.

DMA and Virtual Memory

Given the use of virtual memory, there is the matter of whether DMA should transfer using virtual addresses or physical addresses. Here are a couple of problems with DMA using physically mapped I/O:

- Transferring a buffer that is larger than one page will cause problems, since the pages in the buffer will not usually be mapped to sequential pages in physical memory.
- Suppose DMA is ongoing between memory and a frame buffer, and the operating system removes some of the pages from memory (or relocates them). The DMA would then be transferring data to or from the wrong page of memory.

One answer is *virtual DMA*. It allows the DMA to use virtual addresses that are mapped to physical addresses during the DMA. Thus, a buffer must be sequential in virtual memory, but the pages can be scattered in physical memory. The operating system could update the address tables of a DMA if a process is moved using virtual DMA, or the operating system could "lock" the pages in memory until the DMA is complete. Figure 6.30 shows address-translation registers added to the DMA device.

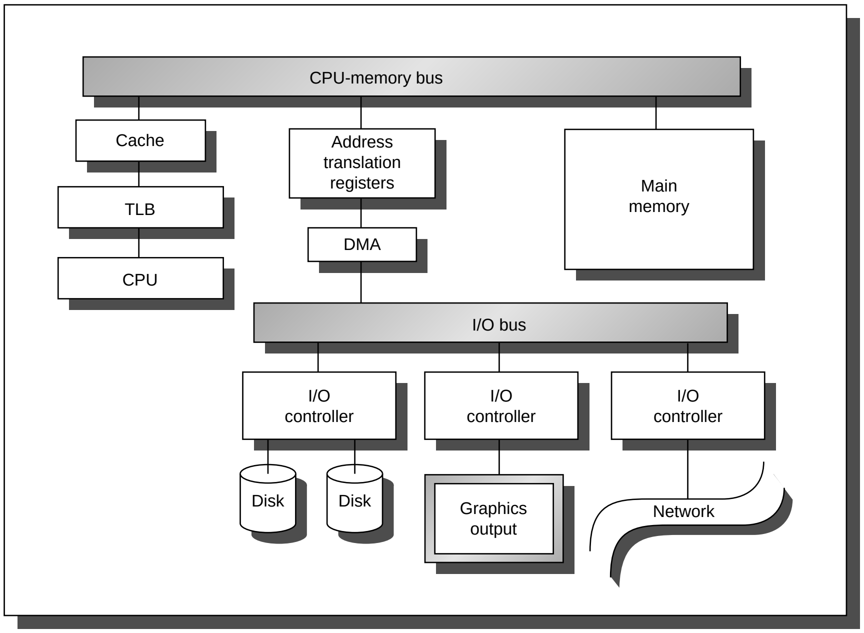


FIGURE 6.30 Virtual DMA requires a register for each page to be transferred in the DMA controller, showing the protection bits and the physical page corresponding to each virtual page.

6.7 Designing an I/O System

The art of I/O is finding a design that meets goals for cost and variety of devices while avoiding bottlenecks to I/O performance. This means that components must be balanced between main memory and the I/O device, because performance—and hence effective cost/performance—can only be as good as the weakest link in the I/O chain. The architect must also plan for expansion so that customers can tailor the I/O to their applications. This expansibility, both in numbers and types of I/O devices, has its costs in longer I/O buses, larger power supplies to support I/O devices, and larger cabinets.

In designing an I/O system, analyze performance, cost, and capacity using varying I/O connection schemes and different numbers of I/O devices of each type. Here is a series of six steps to follow in designing an I/O system. The answers for each may be dictated by market requirements or simply by cost/performance goals.

1. List the different types of I/O devices to be connected to the machine, or list the standard buses that the machine will support.
2. List the physical requirements for each I/O device. This includes volume, power, connectors, bus slots, expansion cabinets, and so on.
3. List the cost of each I/O device, including the portion of cost of any controller needed for this device.
4. Record the CPU resource demands of each I/O device. This should include
   - Clock cycles for instructions used to initiate an I/O, to support operation of an I/O device (such as handling interrupts), and complete I/O
   - CPU clock stalls due to waiting for I/O to finish using the memory, bus, or cache
   - CPU clock cycles to recover from an I/O activity, such as a cache flush
5. List the memory and I/O bus resource demands of each I/O device. Even when the CPU is not using memory, the bandwidth of main memory and the I/O bus is limited.
6. The final step is assessing the performance of the different ways to organize these I/O devices. Performance can only be properly evaluated with simulation, though it may be estimated using queuing theory.

You then select the best organization, given your performance and cost goals.

Cost/performance goals affect the selection of the I/O scheme and physical design. Performance can be measured either as megabytes per second or I/Os per second, depending on the needs of the application. For high performance, the only limits should be speed of I/O devices, number of I/O devices, and speed of memory and CPU. For low cost, the only expenses should be those for the I/O devices themselves and for cabling to the CPU. Cost/performance design, of course, tries for the best of both worlds.

To make these ideas clearer, let's go through several examples.

EXAMPLE First, let's look at the impact on the CPU of reading a disk page directly into the cache. Make the following assumptions:

- Each page is 16 KB, and the cache-block size is 64 bytes.
- The addresses corresponding to the new page are *not* in the cache.
- The CPU will not access any of the data in the new page.
- 95% of the blocks that were displaced from the cache will be read in again, and each will cause a miss.
- The cache uses write back, and 50% of the blocks are dirty on average.
- The I/O system buffers a full cache block before writing to the cache (this is called a *speed-matching buffer*, matching transfer bandwidth of the I/O system and memory).
- The accesses and misses are spread uniformly to all cache blocks.
- There is no other interference between the CPU and I/O for the cache slots.
- There are 15,000 misses every 1 million clock cycles when there is *no* I/O.
- The miss penalty is 30 clock cycles, plus 30 more cycles to write the block if it was dirty.

Assuming one page is brought in every 1 million clock cycles, what is the impact on performance?

ANSWER Each page fills 16,384/64 or 256 blocks. I/O transfers do not cause cache misses on their own because entire cache blocks are transferred. However, they do displace blocks already in the cache. If half of the displaced blocks are dirty, it takes 128×30 clock cycles to write them back to memory. There are also misses from 95% of the blocks displaced in the cache because they are referenced later, adding another $95\% \times 256$, or 244 misses. Since this data was placed into the cache from the I/O system, all these blocks are dirty and will need to be written back when replaced. Thus, the total is on average $128 \times 30 + 244 \times 60$ more clock cycles than the original 1,000,000 + 7500 × 30 + 7500 × 60. This turns into a 1% decrease in performance:

$$\frac{128 \times 30 + 244 \times 60}{1{,}000{,}000 + 7500 \times 30 + 7500 \times 60} = \frac{18{,}480}{1{,}675{,}000} = 0.011$$

■

Now let's take a long look at the cost/performance of different I/O organizations. A simple way to perform this analysis is to look at maximum throughput, assuming that resources can be used at 100% of their maximum rate without side effects from interference. (A later example takes a more realistic view.)

EXAMPLE Assume the following performance and cost information:

- A 500-MIPS CPU costing $30,000.
- A 16-byte-wide memory with a 100-ns cycle time.
- 200 MB/sec I/O bus with room for 20 SCSI-2 buses and controllers.
- SCSI-2 buses that can transfer 20 MB/sec and support up to 15 disks per bus (these are also called SCSI *strings*).
- A $1500 SCSI-2 controller that adds 1 ms of overhead to perform a disk I/O.
- An operating system that uses 10,000 CPU instructions for a disk I/O.
- A choice of a large disk containing 8 GB or a small disk containing 2 GB, each costing $0.25 per MB.
- Both disks rotate at 7200 RPM, have an 8-ms average seek time, and can transfer 6 MB/sec.
- The storage capacity must be 200 GB.
- The average I/O size is 16 KB.

Evaluate the cost per I/O per second (IOPS) of using small or large drives. Assume that every disk I/O requires an average seek and average rotational delay. Use the optimistic assumption that all devices can be used at 100% of capacity and that the workload is evenly divided among all disks.

ANSWER I/O performance is limited by the weakest link in the chain, so we evaluate the maximum performance of each link in the I/O chain for each organization to determine the maximum performance of that organization.

Let's start by calculating the maximum number of IOPS for the CPU, main memory, and I/O bus. The CPU I/O performance is determined by the speed of the CPU and the number of instructions to perform a disk I/O:

$$\text{Maximum IOPS for CPU} = \frac{\text{500 MIPS}}{\text{10,000 instructions per I/O}} = 50{,}000$$

The maximum performance of the memory system is determined by the memory cycle time, the width of the memory, and the size of the I/O transfers:

$$\text{Maximum IOPS for main memory} = \frac{(1/100\text{ ns}) \times 16}{\text{16 KB per I/O}} \approx 10{,}000$$

The I/O bus maximum performance is limited by the bus bandwidth and the size of the I/O:

$$\text{Maximum IOPS for the I/O bus} = \frac{\text{200 MB/sec}}{\text{16 KB per I/O}} \approx 12{,}500$$

Thus, no matter which disk is selected, the CPU and main memory limit the maximum performance to no more than 10,000 IOPS.

Now it's time to look at the performance of the next link in the I/O chain, the SCSI-2 controllers. The time to transfer 16 KB over the SCSI-2 bus is

$$\text{SCSI-2 bus transfer time} = \frac{\text{16 KB}}{\text{20 MB/sec}} = 0.8\text{ ms}$$

Adding the 1-ms SCSI-2 controller overhead means 1.8 ms per I/O, making the maximum rate per controller

$$\text{Maximum IOPS per SCSI-2 controller} = \frac{1}{1.8\text{ ms}} = 556\text{ IOPS}$$

All the organizations will use several controllers, so 556 IOPS is not the limit for the whole system.

The final link in the chain is the disks themselves. The time for an average disk I/O is

$$\text{I/O time} = 8\text{ ms} + \frac{0.5}{\text{7200 RPM}} + \frac{\text{16 KB}}{\text{6 MB/sec}} = 8 + 4.2 + 2.7 = 14.9\text{ ms}$$

so the disk performance is

$$\text{Maximum IOPS (using average seeks) per disk} = \frac{1}{14.9\text{ ms}} \approx 67\text{ IOPS}$$

The number of disks in each organization depends on the size of each disk: 200 GB can be either 25 8-GB disks or 100 2-GB disks. The maximum number of I/Os for all the disks is

$$\text{Maximum IOPS for 25 8-GB disks} = 25 \times 67 = 1675$$
$$\text{Maximum IOPS for 100 2-GB disks} = 100 \times 67 = 6700$$

Thus, provided there are enough SCSI-2 strings, the disks become the new limit to maximum performance: 1675 IOPS for the 8-GB disks and 6700 for the 2-GB disks.

Although we have determined the performance of each link of the I/O chain, we still have to determine how many SCSI-2 buses and controllers to use and how many disks to connect to each controller, as this may further limit maximum performance. The I/O bus is limited to 20 SCSI-2 controllers, and the limit is 15 disks per SCSI-2 string. The minimum number of controllers for the 8-GB disks is

$$\text{Minimum number of SCSI-2 strings for 25 8-GB disks} = \left\lceil \frac{25}{15} \right\rceil \text{ or } 2$$

and for 2-GB disks

$$\text{Minimum number of SCSI-2 strings for 100 2-GB disks} = \left\lceil \frac{100}{15} \right\rceil \text{ or } 7$$

We can calculate the maximum IOPS for each configuration:

$$\text{Maximum IOPS for 2 SCSI-2 strings} = 2 \times 556 = 1112$$
$$\text{Maximum IOPS for 7 SCSI-2 strings} = 7 \times 556 = 3892$$

The maximum performance of this number of controllers is slightly lower than the disk I/O throughput, so let's also calculate the number of controllers so they don't become a bottleneck. One way is to find the number of disks they can support per string:

$$\text{Number of disks per SCSI string at full bandwidth} = \left\lfloor \frac{556}{67} \right\rfloor = \lfloor 8.3 \rfloor \text{ or } 8$$

and then calculate the number of strings:

$$\text{Number of SCSI strings for full bandwidth 8-GB disks} = \left\lceil \frac{25}{8} \right\rceil = \lceil 3.1 \rceil \text{ or } 4$$

$$\text{Number of SCSI strings for full bandwidth 2-GB disks} = \left\lceil \frac{100}{8} \right\rceil = \lceil 12.5 \rceil \text{ or } 13$$

This establishes the performance of four organizations: 25 8-GB disks with 2 or 4 SCSI-2 strings and 100 2-GB disks with 7 or 13 SCSI-2 strings. Using the format

$$\text{Min(CPU limit, memory limit, I/O bus limit, disk limit, string limit)}$$

the maximum performance of each option is limited by the bottleneck (in boldface):

8-GB disks, 2 strings	= Min(50,000, 10,000, 12,500, 1675,**1112**)	= 1112 IOPS
8-GB disks, 4 strings	= Min(50,000, 10,000, 12,500, **1675**, 2224)	= 1675 IOPS
2-GB disks, 7 strings	= Min(50,000, 10,000, 12,500, 6700, **3892**)	= 3892 IOPS
2-GB disks, 13 strings	= Min(50,000, 10,000, 12,500, **6700**, 7228)	= 6700 IOPS

We can now calculate the cost for each organization:

8-GB disks, 2 strings	= \$30,000 + 2 × \$1500 + 25 × (8192 × \$0.25)	= \$84,200
8-GB disks, 4 strings	= \$30,000 + 4 × \$1500 + 25 × (8192 × \$0.25)	= \$87,200
2-GB disks, 7 strings	= \$30,000 + 7 × \$1500 + 100 × (2048 × \$0.25)	= \$91,700
2-GB disks, 13 strings	= \$30,000 + 13 × \$1500 + 100 × (2048 × \$0.25)	=\$100,700

Finally, the cost per IOPS for each of the four configurations is \$76, \$52, \$24, and \$15, respectively. Calculating the maximum number of average I/Os per second, assuming 100% utilization of the critical resources, the best cost/performance is the organization with the small disks and the largest number of controllers. The small disks have about 3.5 times better cost/performance than the large disks in this example. The only drawback is that the larger number of disks will affect system availability unless some form of redundancy is added (see section 6.5). ■

This example assumed that resources can be used 100%. It is instructive to see what the bottleneck is in each organization.

EXAMPLE For the organizations in the last example, calculate the percentage of utilization of each resource in the computer system.

ANSWER Figure 6.31 gives the answer. Either the disks or the SCSI buses are the bottleneck.

Resource	8-GB disks, 2 strings	8-GB disks, 4 strings	2-GB disks, 7 strings	2-GB disks, 13 strings
CPU	2%	3%	8%	13%
Memory	11%	17%	39%	67%
I/O bus	9%	13%	31%	54%
SCSI-2 buses	100%	75%	100%	93%
Disks	66%	100%	58%	100%
IOPS	1112	1675	3892	6700

FIGURE 6.31 The percentage of utilization of each resource and peak IOPS given the four organizations in the previous example. Either the SCSI-2 buses or the disks are the bottleneck.

■

While it is useful to learn where the bottleneck is, it's more important to see the impact on response time as we approach 100% utilization of a resource. Let's do this for one configuration from Figure 6.31.

EXAMPLE Recalculate performance for, say, the second column in Figure 6.31, but this time in terms of response time. Assume that all requests are in a single wait line. To simplify the calculation, ignore the SCSI-2 strings and just calculate for the 25 disks. According to Figure 6.31, the peak I/O rate is 1675 IOPS. Plot the mean response time for the following number of I/Os per second: 1000, 1100, 1200, 1300, 1400, 1500, 1550, 1600, 1625, 1650, 1670. Assume the time between requests is exponentially distributed.

ANSWER To be able to calculate the average response time, we need the equation for an M/M/m queue; that is, for m servers rather than one. From Jain [1991] we get the formulas for that queue:

$$\text{Server utilization} = \frac{\text{Arrival rate}}{\dfrac{1}{\text{Time}_{\text{server}}/\text{m}}} = \text{Arrival rate} \times \frac{\text{Time}_{\text{server}}}{\text{m}}$$

$$\text{Time}_{\text{system}} = \text{Time}_{\text{server}} \times \left(1 + \frac{\text{Prob}\ (>=\text{m})}{\text{m} \times 1 - \text{Util}_{\text{server}}}\right)$$

That is, the average service time for m servers is simply the average service time of one server divided by the number of servers.

From the example above we know that we have 25 disks and that the mean service time is 14.9 ms. Figure 6.32 shows the utilization and mean response time for each of the request rates, and Figure 6.33 plots the response times as the request rate varies.

Request rate	Utilization	Mean response time (ms)
1000	60%	15.8
1100	66%	16.0
1200	72%	16.4
1300	77%	16.9
1400	83%	17.9
1500	89%	19.9
1550	92%	22.1
1600	95%	27.1
1625	97%	33.1
1650	98%	49.8
1670	100%	137.0

FIGURE 6.32 Utilization and mean response time for 25 large disks in the prior example, ignoring the impact of SCSI-2 buses and controllers.

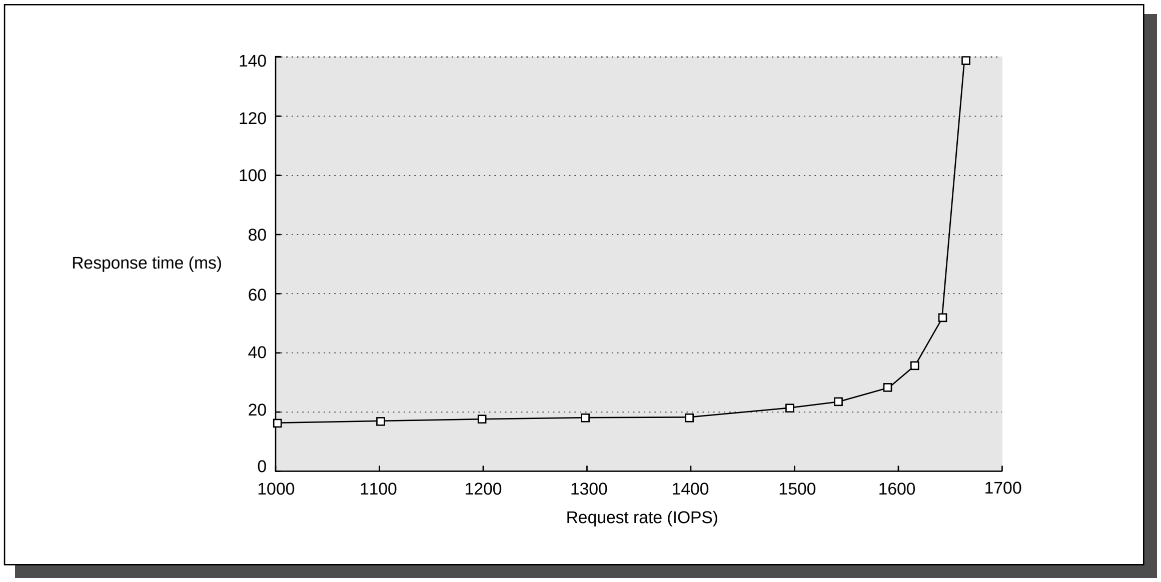


FIGURE 6.33 X-Y plot of response times in Figure 6.32.

■

Figure 6.33 shows the severe increase in response time when trying to use 100% of a server. A variety of rules of thumb have been evolved to guide I/O designers to keep response time and contention low:

- No I/O bus should be utilized more than 75%.
- No disk string should be utilized more than 40%.
- No disk arm should be seeking more than 60% of the time.
- No disk should be used more than 80% of the time.

EXAMPLE Recalculate performance in the example above using these rules of thumb, and show the utilization of each component.

ANSWER Figure 6.31 shows that the I/O bus is far below the suggested guidelines, so we concentrate on the disks, utilization of disk seeking, and SCSI-2 bus. The new limit on IOPS for disks used 80% of the time is 67 × 0.8 = 54 IOPS. The utilization of seek time per disk is

$$\frac{\text{Time of average seek}}{\text{Time between I/Os}} = \frac{8}{\dfrac{1}{54\text{ IOPS}}} = \frac{8}{18.5} = 43\%$$

which is below the rule of thumb. The biggest impact is on the SCSI-2 bus:

$$\text{Suggested IOPS per SCSI-2 string} = \frac{1}{1.8\text{ ms}} \times 40\% = 222\text{ IOPS}$$

With this data we can recalculate IOPS for each organization:

8-GB disks, 2 strings	= Min(50,000,10,000, 9375, 1350, **444**)	= 444 IOPS
8-GB disks, 4 strings	= Min(50,000,10,000, 9375, 1350, **888**)	= 888 IOPS
2-GB disks, 7 strings	= Min(50,000,10,000, 9375, 5400, **1554**)	= 1554 IOPS
2-GB disks, 13 strings	= Min(50,000,10,000, 9375, 5400, **2886**)	= 2886 IOPS

Under these assumptions, the small disks have about 3.5 times the performance of the large disks.

Clearly, the string bandwidth is the bottleneck now. The number of disks per string that would not exceed the guideline is

$$\text{Number of disks per SCSI-2 string at full bandwidth} = \left\lfloor \frac{222}{54} \right\rfloor = \lfloor 4.1 \rfloor = 4$$

and the ideal number of strings is

$$\text{Number of SCSI-2 strings with 8-GB disks} = \left\lceil \frac{25}{4} \right\rceil = \lceil 6.3 \rceil = 7$$

$$\text{Number of SCSI-2 strings for full bandwidth with 2-GB disks} = \left\lceil \frac{100}{4} \right\rceil = 25$$

This suggestion is fine for 8-GB disks, but the I/O bus is limited to 20 SCSI-2 controllers and strings so that becomes the limit for 2-GB disks:

8-GB disks, 7 strings	= Min(50,000, 10,000, 9375, **1350**, 1554)	= 1350 IOPS
2-GB disks, 20 strings	= Min(50,000, 10,000, 9375, 5400, **4440**)	= 4440 IOPS

Notice that the IOPS for the large disks is in the flat part of the response time graph in Figure 6.33, as we would hope. We can now calculate the cost for each organization:

8-GB disks, 7 strings	= \$30,000 + 7 × \$1500 + 25 × (8192 × \$0.25)	= \$91,700
2-GB disks, 20 strings	= \$30,000 + 20 × \$1500 + 100 × (2048 × \$0.25)	= \$111,200

The respective cost per IOPS is \$68 versus \$25, or an advantage of about 2.7 for the small disks. Compared with the earlier naive assumption that we could use 100% of resources, the cost per IOPS increased \$10 to \$15. Figure 6.34 shows the new utilization of each resource by following these guidelines. Following the rule of thumb of 40% string utilization sets the performance limit in every case. Exercise 6.18 explores what happens when this SCSI limit is relaxed.

Resource	**8-GB disks, 2 strings**	**8-GB disks, 4 strings**	**2-GB disks, 7 strings**	**2-GB disks, 13 strings**	**8-GB disks, 7 strings**	**2-GB disks, 20 strings**
CPU	1%	2%	3%	6%	3%	9%
Memory	4%	9%	16%	29%	16%	44%
I/O bus	4%	7%	12%	23%	12%	36%
SCSI-2 buses	40%	40%	40%	40%	40%	40%
Disks	27%	53%	23%	43%	23%	66%
Seek utilization	14%	28%	12%	23%	12%	36%
IOPS	444	888	1554	2886	1350	4400

FIGURE 6.34 The percentage of utilization of each resource, given the six organizations in this Example, which tries to limit utilization of key resources to the rules of thumb given above.

■

Queuing theory can also help us to answer questions about I/O controllers and buses.

EXAMPLE The SCSI controller will send requests down the bus to the device and then get data back on a read. One issue is the impact of returning the data in a single 16-KB transfer versus four 4-KB transfers. How long does it take for the drive to see the request for each workload? Assume that there are many disks on the SCSI bus, that the time between arriving SCSI requests is exponential, that the bus is occupied during the entire transfer, that the overhead for each SCSI activity is 1 ms plus the time to transfer the data, and that the CPU issues 100 disk reads per second on this SCSI bus.

ANSWER The times between arrivals are exponential, but we need the distribution of the service times on the SCSI bus. For the 16-KB transfer size there are just two sizes: very small and 16 KB, and so the times are 1 ms or

$$1\text{ ms} + \frac{16\text{ KB}}{20\text{ MB/sec}} = 1 + 0.8 = 1.8\text{ ms}$$

In fact, for each CPU request taking 1 ms there is exactly one transfer taking 1.8 ms, so the distribution is half 1-ms service times and half 1.8-ms service times. A 4-KB transfer takes

$$1\text{ ms} + \frac{4\text{ KB}}{20\text{ MB/sec}} = 1 + 0.2 = 1.2\text{ ms}$$

For every request of 1.0 ms there are four 1.2-ms transfers. Neither distribution is exponential, so we must use the general model for service interarrival times. Since the SCSI bus acts as a single queue following the FIFO discipline, we must use the M/G/1 model to answer this question.

The proper formula to predict the time before a single transfer comes from page 513:

$$\text{Time}_{\text{queue}} = \frac{\text{Time}_{\text{server}} \times (1 + \text{C}) \times \text{Server utilization}}{2 \times (1 - \text{Server utilization})}$$

Thus we must first calculate $\text{Time}_{\text{server}}$, Server utilization, and C.

For a single transfer, the average time before the disk can transfer is

$$\text{Time}_{\text{server}} = \text{Weighted mean time} = \frac{f_1 \times T_1 + f_2 \times T_2 + \ldots + f_n \times T_n}{f_1 + f_2 + \ldots + f_n} = \frac{0.5 \times 1 + 0.5 \times 1.8}{0.5 + 0.5} = 1.4 \text{ ms}$$

$$\text{Server utilization} = \frac{\text{Arrival rate}}{1/\text{Time}_{\text{server}}} = \text{Arrival rate} \times \text{Time}_{\text{server}} = (100 \times (1+1))/\text{sec} \times 1.4 \text{ ms} = 200/\text{sec} \times 0.0014 \text{ sec} = 0.28$$

$$\text{Variance} = \frac{f_1 \times T_1^2 + f_2 \times T_2^2 + \ldots + f_n \times T_n^2}{f_1 + f_2 + \ldots + f_n} - \text{Time}_{\text{server}}^2$$

$$\text{Variance} = \frac{0.5 \times 1^2 + 0.5 \times 1.8^2}{0.5 + 0.5} - 1.4^2 = 0.5 + 1.62 - 1.96 = 2.12 - 1.96 = 0.16$$

$$\text{C} = \frac{\text{Variance}}{\text{Time}_{\text{server}}^2} = \frac{0.16}{1.4^2} = \frac{0.16}{1.96} = 0.082$$

$$\text{Time}_{\text{queue}} = \frac{\text{Time}_{\text{server}} \times (1 + \text{C}) \times \text{Server utilization}}{2 \times (1 - \text{Server utilization})} = \frac{1.4 \text{ ms} \times (1 + 0.082) \times 0.28}{2 \times (1 - 0.16)} = \frac{0.424}{1.440} \text{ ms} = 0.294 \text{ ms}$$

For the case where the transfer is broken into four 4-KB pieces, the time is

$$\text{Time}_{\text{server}} = \text{Weighted mean time} = \frac{f_1 \times T_1 + f_2 \times T_2 + \ldots + f_n \times T_n}{f_1 + f_2 + \ldots + f_n} = \frac{0.2 \times 1 + 0.8 \times 1.2}{0.2 + 0.8} = 1.16 \text{ ms}$$

$$\text{Server utilization} = \text{Arrival rate} \times \text{Time}_{\text{server}} = (100 \times (1+4))/\text{sec} \times 1.16 \text{ ms} = 500/\text{sec} \times 0.0016 \text{ sec} = 0.58$$

$$\text{Variance} = \frac{f_1 \times T_1^2 + f_2 \times T_2^2 + \ldots + f_n \times T_n^2}{f_1 + f_2 + \ldots + f_n} - \text{Time}_{\text{server}}^2$$

$$\text{Variance} = \frac{0.2 \times 1^2 + 0.8 \times 1.2^2}{0.2 + 0.8} - 1.16^2 = 0.2 + 1.152 - 1.346 = 1.352 - 1.346 = 0.006$$

$$C = \frac{\text{Variance}}{\text{Time}_{\text{server}}^2} = \frac{0.006}{1.16^2} = \frac{0.006}{1.346} = 0.005$$

$$\text{Time}_{\text{queue}} = \frac{\text{Time}_{\text{server}} \times (1 + C) \times \text{Server utilization}}{2 \times (1 - \text{Server utilization})} = \frac{1.16 \text{ ms} \times (1 + 0.005) \times 0.58}{2 \times (1 - 0.58)} = \frac{0.676}{0.840} \text{ ms} = 0.805 \text{ ms}$$

For these parameters, the single large transfer wins: 0.3 ms versus 0.8 ms. Although it might seem better to break the transfer into smaller pieces so that a request doesn't have to wait for the long transfer, the collective SCSI overhead on each transfer increases bus utilization so as to overcome the benefits of the shorter transfers. ■

6.8 Putting It All Together: UNIX File System Performance

This section compares the file-system performance of several operating systems and hardware systems in use in 1995 (see Figure 6.35). It is based on a paper by Chen and Patterson [1994a].

As a preview, this evaluation once again shows that I/O performance is limited by the weakest link in the chain between the disk and the operating system. The hardware determines potential I/O performance, but the operating system determines how much of that potential is delivered. In particular, for UNIX systems the file cache is critical to I/O performance. The main observations are that file cache performance of UNIX on mainframes and mini-supercomputers is no better than workstations, and that file caching policy is of overriding importance. Optimized memory systems can increase read performance, but the operating-system policy on writes can result in orders of magnitude differences in file cache performance.

Computer	Alpha AXP/ 3000	Dec-Station 5000	Dec-Station 5000	HP 730	IBM RS/ 6000	Sun Sparc-Station 1	Sun Sparc-Station 10	Convex C2	IBM 3090
Operating system	OSF-1 1.3	Sprite LFS	Ultrix 4.2A 47	HP/UX 8.07	AIX 3.1.5	SunOS 4.1	Solaris 2.1	Convex-OS 10.1	AIX/ ESA on VM
Processor model	400	200	200	730	550	1+	30	C20	600J VF
Year proc. shipped	1993	1990	1990	1991	1991	1989	1992	1988	1990
Approx. $ as tested	$30K	$20K	$15K	$25K	$30K	$15K	$20K	$750K	$1000K
Proc. clock rate (MHz)	133	25	25	66	41.7	25	33	25	69
Proc. perf. SPECint92	75	19	19	48	34	12	45	≈ 10–20	≈ 35–45
Cache size (levels 1 & 2 in KB)	L1: 8,8 L2: 512	L1: 64,64	L1: 64,64	L1: 128,256	L1: 8,64	L1: 64	L1: 20,16 L2: 1024	L1: 8,4	
Memory size (MB)	64	32	32	32	64	28	128	1024	128 VM partition
Memory perf. (MB/sec)	300	100	100	264	222	80	88	200	
I/O bus	Turbo-channel	SCSI-I	SCSI-I	Fast SCSI-II	SCSI	SCSI-I	SCSI-I	IPI-2	IBM Channel
Disk(s)	1 SCSI DEC RZ26	3 CDC Wren (RAID 0)	1 DEC RZ55	1 HP 1350SX	1 IBM 93x 2355	1 CDC Wren IV	1 Segate Elite (5400 RPM)	4 DKD-502 (RAID 5)	1 IBM 3390

FIGURE 6.35 Machines and operating systems evaluated in this section. Note that the oldest machine is the Convex C20, which first shipped in 1988. AIX/ESA is run under VM because there are not enough people at that installation to justify running it native. For cache parameters, the first level 1 (L1) number is the instruction cache size and the second L1 number is the data cache size; a single number means a unified cache.

Disk Subsystem Performance

A comprehensive evaluation of disk performance is problematic. I/O performance is limited by the slowest component between memory and disks: It can be the main memory, the CPU-memory bus, the bus adapter, the I/O bus, the I/O controller, or the disks themselves. The trend toward open systems exacerbates the complexity of measuring I/O performance, for a particular machine can be configured with many different kinds of disks, disk controllers, and even I/O buses. In addition to the hardware components, the policies of the operating system affect I/O performance of a workload. The number of combinations is staggering.

If we were interested in comparing the I/O performance of hardware-software systems, then ideally we would use many of the same components to reduce the number of variables. This ideal has several practical obstacles. First, few workstations share the same operating system, CPU, or CPU-memory bus, so they may be unique to each machine. And a different CPU-memory bus requires a different bus adaptor. This leaves the I/O bus, I/O controller, and disks to be potentially in common. The problem now is that there is no standard configuration of these components across manufacturers, so it is unlikely that customers would normally buy the same configuration from different manufacturers. This leaves the evaluator the unattractive alternative of purchasing computer systems with common I/O subsystems simply to evaluate performance; few organizations have the budgets for such an effort.

We can now present the results in proper context. Figure 6.36 shows disk performance when reading for the machines in Figure 6.35. The Convex minisupercomputer, with the RAID of four disks and the fast IPI-2 I/O bus, is at the top of the chart; the SparcStation 10 is second because of its fast single disk. The 3090 mainframe, with its single 3390 disk, comes in a surprisingly low sixth place.

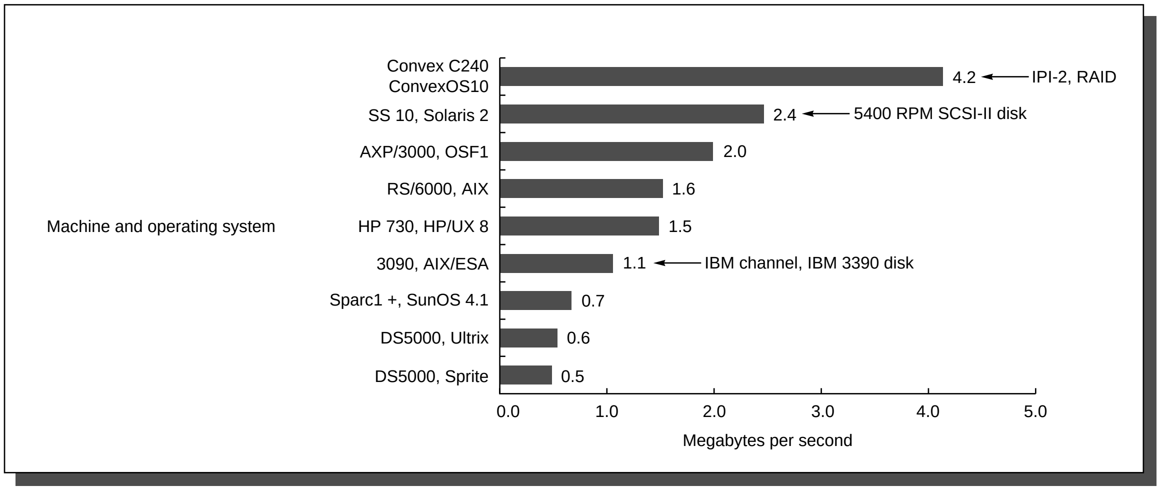


FIGURE 6.36 Disk performance for the machines in Figure 6.35. These results are for reads of size 32 KB. Except where otherwise noted in the figure, the machines use SCSI I/O buses, SCSI controllers, and SCSI disks that rotate at 3600 RPM. Read performance is from 1.5 to 2.8 times faster than write performance, except for the Sprite system. Sprite's Log-Structured File System is optimized for writes, which are 1.3 times faster than reads on the DS 5000. (Section 6.4 explains the measurement method of data collection; we started with the measured 100% read performance at nominal access sizes and then interpolated to determine performance for a common 32-KB access size. The only numbers adjusted by more than 5% were for the Alpha AXP/3000, DS 5000/ Ultrix, and the Convex.)

Given the warnings above, we *cannot* say that IBM mainframes have lower disk performance than workstations, nor that Convex has the fastest disk subsystem. We *can* say that the IBM 3090-600J running AIX/ESA under VM performs 32-KB reads to a single IBM 3390 disk drive much more slowly than a Convex C240 running Convex OS10 reads 32-KB blocks from a four-disk RAID.

The conclusions we draw from Figure 6.36 are that many workstation I/O subsystems can sustain the performance of a high-speed single disk, that a RAID disk array can deliver much higher performance, and that the performance of a single mainframe disk on a 3090 model 600J running AIX/ESA under VM is no faster than many workstations.

Basic File Cache Performance

For UNIX systems the most important factor in I/O performance is not how fast the disk is, or how efficiently it is used, but *whether* it is used. Operating systems designers' concern for performance led them to cache-like optimizations, using main memory as a "cache" for disk traffic to improve I/O performance. Since main memory is much faster than disks, file caches yield a substantial performance improvement and are found in every UNIX operating system. Figure 6.37 shows the change in disk I/Os versus a cacheless system measured as miss rate.

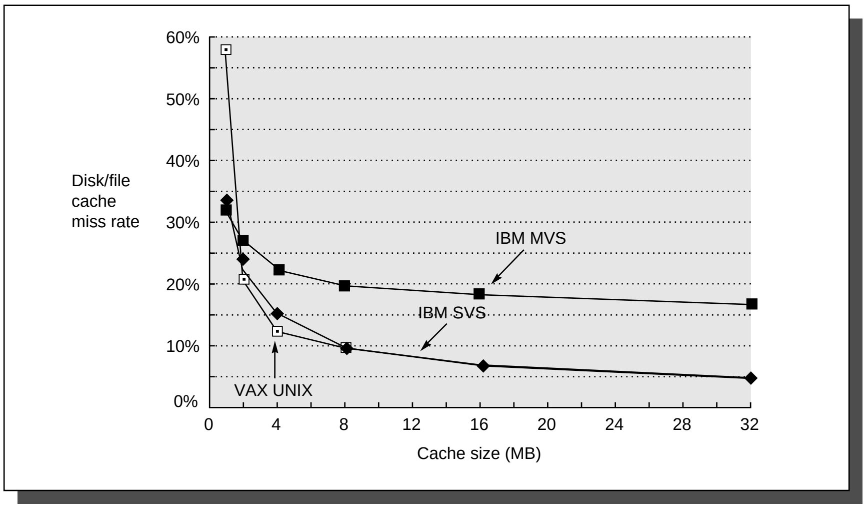


FIGURE 6.37 The effectiveness of a file cache or disk cache on reducing disk I/Os versus cache size. Ousterhout et al. [1985] collected the VAX UNIX data on VAX-11/785s with 8 MB to 16 MB of main memory, running 4.2 BSD UNIX using a 16-KB block size. Smith [1985] collected the IBM SVS and IBM MVS traces on IBM 370/168 using a one-track block size (which varied from 7294 bytes to 19,254 bytes, depending on the disk). The difference between a file cache and a disk cache is that the file cache uses logical block numbers while a disk cache uses addresses that have been mapped to the physical sector and track on a disk. This difference is similar to the difference between a virtually addressed and a physically addressed cache (see section 5.5).

Figure 6.38 shows this file cache performance for the machines of Figure 6.35. The first thing to notice is the change in scale of the chart: machines read from their file caches 3 to 25 times faster than from their disks. The performance of the file cache is determined by the processor, cache, CPU-memory bus, main memory, and operating system. Except for the size of memory and perhaps the operating system, there is little choice in these components when selecting a computer. Hence observations about commercial systems can be drawn about file cache performance with much more confidence than with the disks, since this portion of the system will be common at most sites.

The biggest surprise is that the mainframe and mini-supercomputers did not lead this chart, given their much greater cost and reputation for high-bandwidth memory systems and CPU-memory buses. At the top of the list is the Alpha AXP/3000 and HP 730 workstation running HP/UX version 8, both delivering over 30 MB per second. Unlike most workstations, the Alpha 3000 and HP 730 have interleaved main memories; that may explain their fast file cache performance. The IBM RS/6000 model 550 comes in third, and it also has high memory bandwidth. This chart also shows rapid file cache improvement in workstations. For example, both the SparcStation 10 and DEC Alpha AXP/3000 are more than four times faster than their predecessors, the SparcStation 1 and the DecStation 5000.

In addition, Figure 6.38 shows the impact of operating systems on I/O performance; the Sprite operating system offers 1.7 times the file cache performance of Ultrix running on the same DecStation 5000 hardware. Sprite does fewer copies when reading data from the file cache than does Ultrix, hence its higher performance. Fewer copies are important for networks as well as storage, as we shall see in the next chapter.

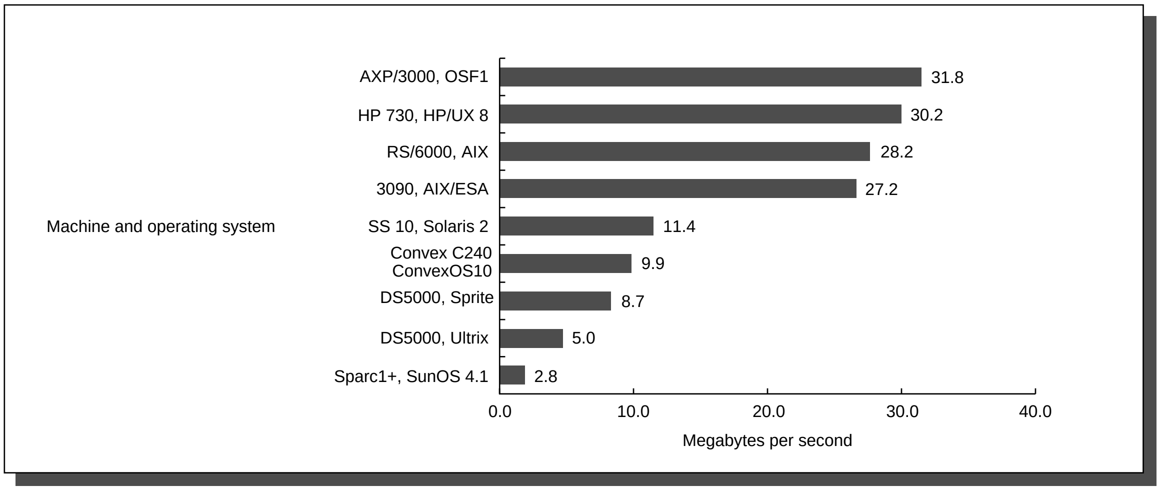


FIGURE 6.38 File cache performance for machines in Figure 6.35. This plot is for 32-KB reads with the number of bytes touched limited to fit within the file cache of each system. Figure 6.39 (page 545) shows the size of the file caches that achieve this performance. (See the caption of Figure 6.39 for details on measurements.)

The Impact of Operating System Policies on File Cache Performance

Given that UNIX systems have common ancestors, we expected that the operating system policies toward I/O would be the same on all machines. Instead, we find that different systems have very different I/O policies, and some policies alter I/O performance by factors of 10 to 75. Even though the machines measured vary significantly in cost, these policies can be more important than the underlying hardware. These file systems are aimed largely at the same customers running the same applications, which calls into question the low performance of some of these policies.

File Cache Size

Since main memory must be used for running programs as well as for the file cache, the first policy decision is how much main memory should be allocated to the file cache. The second is whether or not the size of the file cache can change dynamically. Early UNIX systems give the file cache a fixed percentage of main memory; this percentage is determined at the time of system generation, and is typically set to 10%. Recent systems allow the barrier between file cache and program memory to vary, allowing file caches to grow to be virtually the full size of the main memory if warranted by the workload. File servers, for example, will surely use much more of their main memory for file cache than will most client workstations.

Figure 6.39 shows these maximum file cache sizes, both in percentage of main memory and in absolute size. The reason for the large variation in percentage of main memory is the file cache size policy. HP/UX version 8, Ultrix, and AIX/ESA all reserve small, fixed portions of main memory for the file cache.

Figure 6.37 (page 542) shows that UNIX workloads benefit from larger file caches, so this fixed-size policy surely hurts I/O performance. Note that Sprite, running on the same hardware as Ultrix, has more than six times the file cache size, and that the SparcStation 1 has a file cache almost as large as the IBM 3090, even though the mainframe has four times the physical memory of the workstation. When the flexible file cache boundary policy is combined with large main memories, we can get astounding file caches: the Convex C240 file cache is almost 900 MB! Thus workloads that would require disk accesses on other machines will instead access main memory on the Convex.

Write Policy

Thus far we have unrealistically left the O out of I/O. There is often some confusion about the definition and implications of alternative write strategies for caches. To lessen that confusion, we first review write policies of processor caches.

Write through with write buffers and write back applies to file caches as well as processor caches. The operating systems community uses the term *asynchronous writes* for writes that allow the processor to continue after updating a write

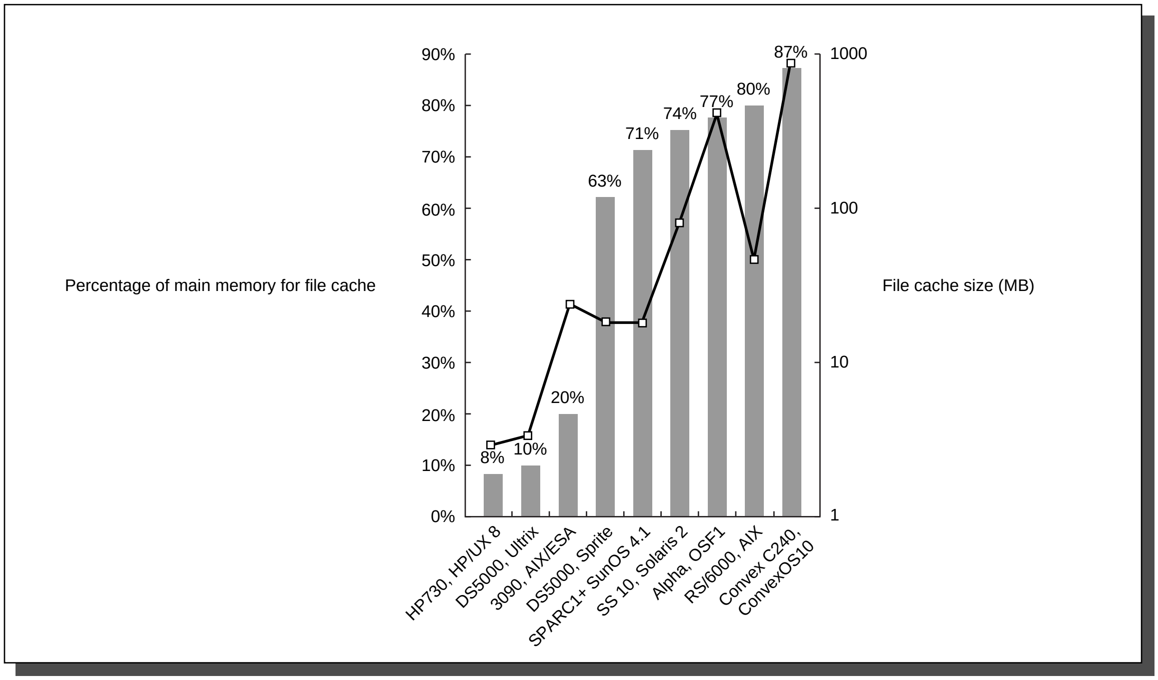


FIGURE 6.39 File cache size. The bar graph shows the maximum percentage of main memory for the file cache, while the line graph shows the maximum size in megabytes, using the log scale on the right. Thus the HP 730 HP/UX version 8 uses only 8% of its 32-MB main memory for its file cache, or just 2.7 MB, and the Convex C240 uses 87% of its 1024-MB main memory, or 890 MB, for its file cache.

buffer. If writes occur infrequently, then this buffer works well; if writes are frequent, then the processor may eventually have to stall until the write buffer empties, limiting the speed to that of the next level of the memory hierarchy. Note that a write buffer does not reduce the number of writes to the next level. It just allows the processor to continue while I/O is in progress, provided that the write buffer is not full.

Figure 6.40 shows the file cache performance as we vary the mix of reads and writes. Clearly HP/UX and Sprite use a write-back cache; while it is hard to see with this figure, the Sun OS 4.1 write performance is nearly as fast as reads and much faster than disks, so it also uses write back. We can tell that the other operating systems use write through because their write performance matches disk speed. Note that three of the highest performers in Figure 6.38—RS/6000, IBM 3090, and Alpha AXP/3000—fall to the back of the pack unless the percentage of reads is more than 90%.

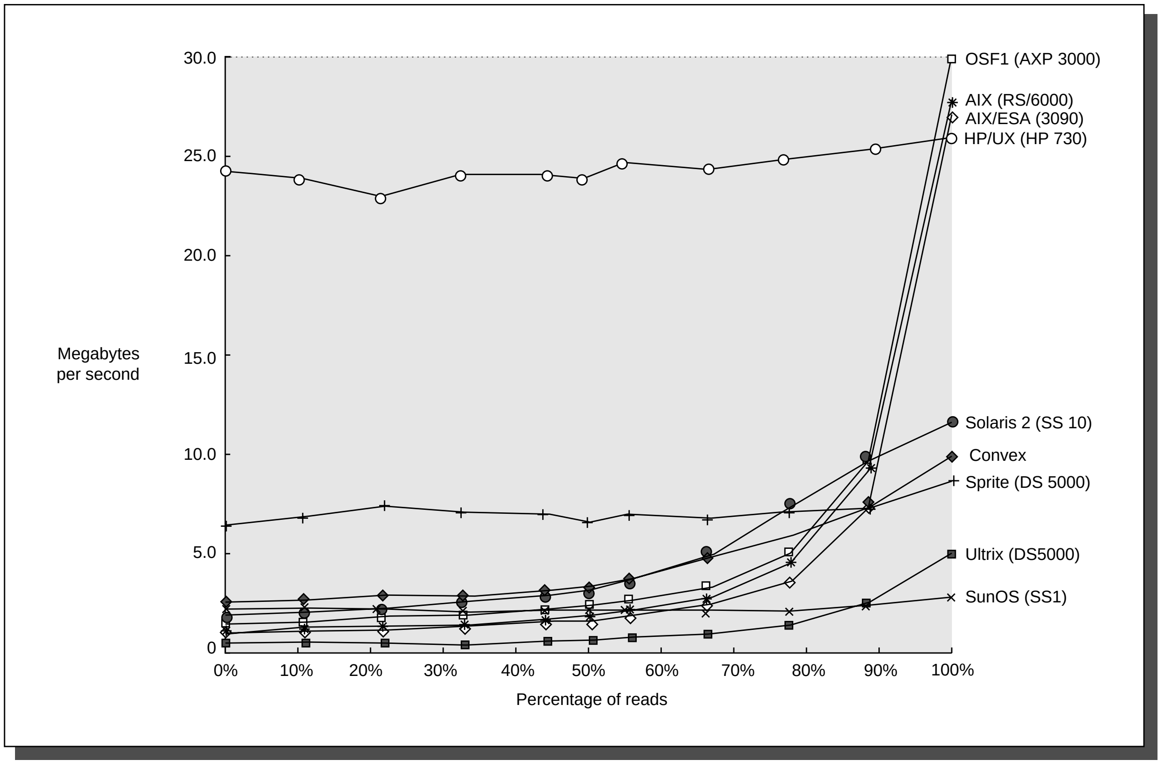


FIGURE 6.40 File cache performance versus read percentage. 0% reads means 100% writes. These accesses all fit within the file caches of the respective machines. Note that the high performance of the file caches of the AXP/3000, RS/6000, and 3090 are only evident for workloads with ≥ 90% reads. Access sizes are 32 KB. (See the caption of Figure 6.36 for details on measurements.)

The effectiveness of caches for writes also depends on the policy of flushing dirty data to the disk; to protect against losing information in case of failures, applications will occasionally issue a command that forces modified data out of the cache and onto the disk. Most UNIX operating systems have a policy of periodically writing dirty data to disk to give a safety window for all applications; typically, the window is 30 seconds.

The short lives of files means that files will be deleted or overwritten and so their data need not be written to disk. Baker et al. [1991] found that this 30-second window captures 65% to 80% of the lifetimes for all files. Hartman and Ousterhout [1993] reported that 36% to 63% of the bytes written do not survive a 30-second window; this number jumps to 60% to 95% in a 1000-second window. Given that such short lifetimes mean that file cache blocks will be rewritten, it seems wise for more operating system policy makers to consider write-back caches provided a 30-second window meets the failure requirements.

Write Policy for Client/Server Computing

Thus far we have been ignoring the network and the possibility that the files exist on a file server. Figure 6.41 shows performance for three combinations of clients and servers:

- An HP 712/60 client running HP-UX 9.05 with an Auspex file server (a 10 SPARC multiprocessor designed for file service)
- An IBM RS6000/520 client running AIX 3.0 with a RS6000/320H server
- A Sun SPARC 10/50 client running Solaris 2.3 with a Sun SPARC 10/50 server

All client/server pairs were connected by an Ethernet local area network. Figure 6.41 shows file cache performance as a percentage of reads when the files are reached over the networks.

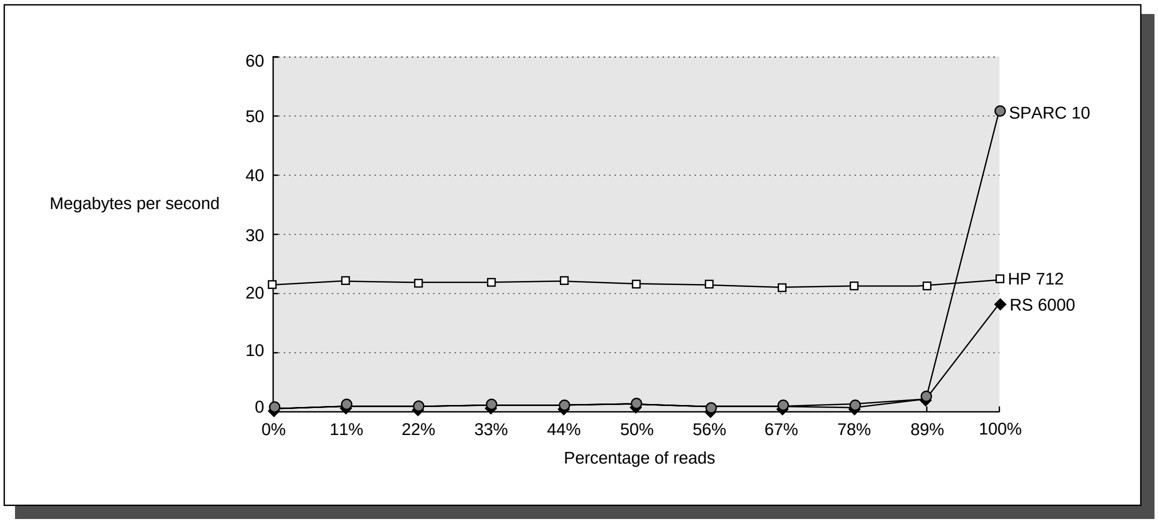


FIGURE 6.41 File cache performance versus percentage of reads for client/server computing. An IBM RS 6000 is 75 times slower than an HP 712 running the DUX network file system.

This experiment brings us to the issue of consistency of files in multiple file caches. The concern is that multiple copies of files in the client caches and on the server create the possibility that someone will access the wrong version of the data. This raises a policy question: How do you ensure that no one accesses stale data? The NFS solution, used by SunOS, is to make all client writes go to the server's disk when the file is closed. It would seem that if the 30-second delay was satisfactory for writes to local disk, then a 30-second delay would also be acceptable before client writes must be sent to the server, thereby allowing the same benefits to accrue. Hence HP/UX 9 offers an alternative network protocol, called

DUX, which allows client-level caching of writes. The server keeps track of potential conflicts and takes the appropriate action only when the file is shared and someone is writing it. Using a shared-bus multiprocessor as a rough analogy to our workstations on the local area network, DUX offers write back with cache coherency, while NFS does write through without write buffers.

The 100%-read case—the rightmost portion of the graph—shows the differences in performance of the hardware, where SPARC 10 is twice as fast as the HP 712. The rest of the graph shows the differences in performance due to the write policies of their operating systems. The HP system is the clear winner: It is 75 times faster than the RS/6000 and 25 times faster than the SPARC 10 for workloads with mostly writes, and still 14 to 20 times faster even when only 20% of the accesses are writes. Put another way, for all but the most heavily read-oriented workloads, the RS/6000 and SPARC 10 clients operate at disk speed while the HP client runs at main memory speed.

Conclusion

Hardware determines the potential I/O performance, but the operating system determines how much of that potential is delivered. As a result of the studies in this section, we conclude the following:

- File caching policy determines performance of most I/O events, and hence is the place to start when trying to improve I/O performance.
- File cache performance in workstations is improving rapidly, with more than fourfold improvements in three years for DEC (AXP/3000 vs. DecStation 5000) and Sun (SparcStation 10 vs. SparcStation 1+).
- File cache performance of UNIX on mainframes and mini-supercomputers is no better than on workstations.
- Workstations can take advantage of high-performance disks.
- RAID systems can deliver much higher disk performance, but cannot overcome weaknesses in file cache policy.

Given the varying decisions in this matter by companies serving the same market, we hope this section motivates file cache designers to give greater emphasis to quantitative evaluations of policy decisions.

6.9 Fallacies and Pitfalls

Pitfall: Comparing the price of media with the price of the packaged system.

This happens most frequently when new memory technologies are compared with magnetic disks. For example, comparing the DRAM-chip price with magnetic-disk packaged price in Figure 6.5 (page 492) suggests the difference is less than a factor of 10, but it’s much greater when the price of packaging DRAM is included. A common mistake with removable media is to compare the media cost not including the drive to read the media. For example, a CD-ROM costs only $2 per gigabyte in 1995, but including the cost of the optical drive may bring the price closer to $200 per gigabyte.

Figure 6.7 (page 495) suggests another example. When comparing a single disk to a tape library, it would seem that tape libraries have little benefit. There are two mistakes in this comparison. The first is that economy of scale applies to tape libraries, and so the economical end is for large tape libraries. The second is that it is more than twice as expensive per gigabyte to purchase a disk storage subsystem that can store terabytes than it is to buy one that can store gigabytes. Reasons for increased cost include packing, interfaces, redundancy to make a system with many disks sufficiently reliable, and so on. These same factors don't apply to tape libraries since they are designed to be sufficiently reliable to store terabytes without extra redundancy. These two mistakes change the ratio by a factor of 10 when comparing large tape libraries with large disk subsystems.

Fallacy: The time of an average seek of a disk in a computer system is the time for a seek of one-third the number of cylinders.

This fallacy comes from confusing the way manufacturers market disks with the expected performance and with the false assumption that seek times are linear in distance. The one-third-distance rule of thumb comes from calculating the distance of a seek from one random location to another random location, not including the current cylinder and assuming there are a large number of cylinders. In the past, manufacturers listed the seek of this distance to offer a consistent basis for comparison. (As mentioned on page 488, today they calculate the "average" by timing all seeks and dividing by the number.) Assuming (incorrectly) that seek time is linear in distance, and using the manufacturer's reported minimum and "average" seek times, a common technique to predict seek time is

$$\text{Time}_{\text{seek}} = \text{Time}_{\text{minimum}} + \frac{\text{Distance}}{\text{Distance}_{\text{average}}} \times (\text{Time}_{\text{average}} - \text{Time}_{\text{minimum}})$$

The fallacy concerning seek time is twofold. First, seek time is *not* linear with distance; the arm must accelerate to overcome inertia, reach its maximum traveling speed, decelerate as it reaches the requested position, and then wait to allow the arm to stop vibrating (*settle time*). Moreover, sometimes the arm must pause to control vibrations. Figure 6.42 plots time versus seek distance for a sample disk. It also shows the error in the simple seek-time formula above. For short seeks, the acceleration phase plays a larger role than the maximum traveling speed, and this phase is typically modeled as the square root of the distance. For disks with more than 200 cylinders, Chen and Lee [1995] modeled the seek distance as

$$\text{Seek time}(\text{Distance}) = a \times \sqrt{\text{Distance} - 1} + b \times (\text{Distance} - 1) + c$$

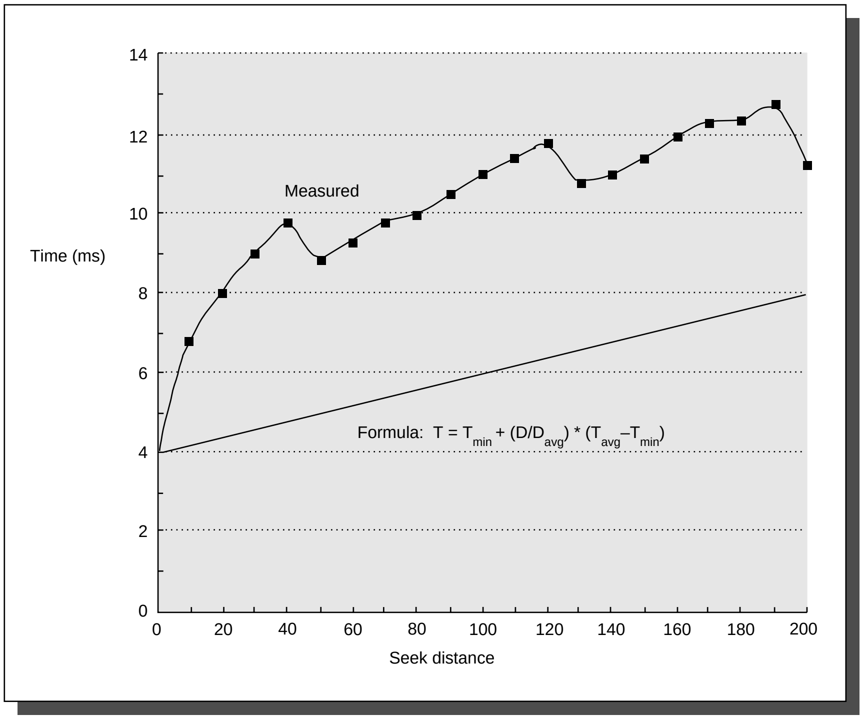
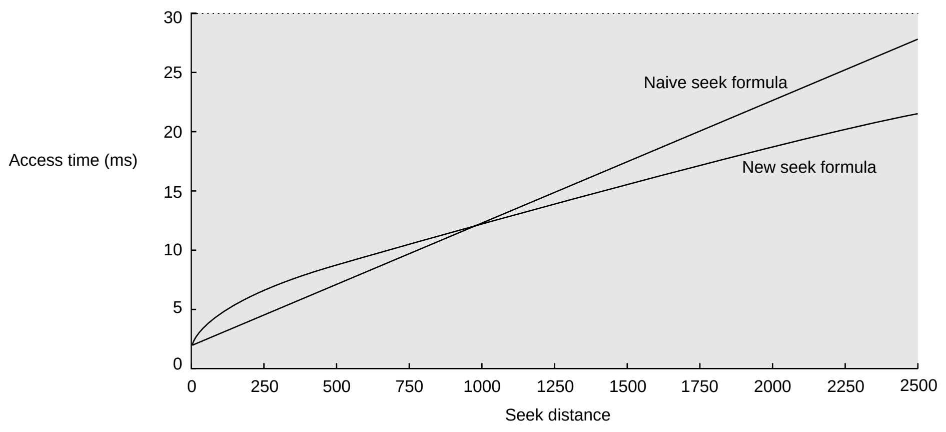


FIGURE 6.42 Seek time versus seek distance for the first 200 cylinders. The Imprimis Sabre 97209 contains 1.2 GB using 1635 cylinders and has the IPI-2 interface [Imprimis 1989]. This is an 8-inch disk. Note that longer seeks can take less time than shorter seeks. For example, a 40-cylinder seek takes almost 10 ms, while a 50-cylinder seek takes less than 9 ms**.**

where *a*, *b*, and *c* are selected for a particular disk so that this formula will match the quoted times for Distance = 1, Distance = max, and Distance = 1/3 max. Figure 6.43 plots this equation versus the fallacy equation for the disk in Figure 6.2.

The second problem is that the average in the product specification would only be true if there was no locality to disk activity. Fortunately, there is both temporal and spatial locality (see page 393 in Chapter 5): disk blocks get used more than once, and disk blocks near the current cylinder are more likely to be used than those farther away. For example, Figure 6.44 shows sample measurements of seek distances for two workloads: a UNIX timesharing workload and a business-processing workload. Notice the high percentage of disk accesses to the same cylinder, labeled distance 0 in the graphs, in both workloads.

Thus, this fallacy couldn't be more misleading. (The Exercises debunk this fallacy in more detail.)



$$a = \frac{-10 \times \text{Time}_{\text{min}} + 15 \times \text{Time}_{\text{avg}} - 5 \times \text{Time}_{\text{max}}}{3 \times \sqrt{\text{Number of cylinders}}} \qquad b = \frac{7 \times \text{Time}_{\text{min}} - 15 \times \text{Time}_{\text{avg}} + 8 \times \text{Time}_{\text{max}}}{3 \times \text{Number of cylinders}} \qquad c = \text{Time}_{\text{min}}$$

FIGURE 6.43 Seek time versus seek distance for sophisticated model versus naive model for the disk in Figure 6.2 (page 490). Chen and Lee [1995] found the equations shown above for parameters a, b, and c worked well for several disks.

Pitfall: Moving functions from the CPU to the I/O processor to improve performance.

There are many examples of this pitfall, although I/O processors can enhance performance. A problem inherent with a family of computers is that the migration of an I/O feature usually changes the instruction set architecture or system architecture in a programmer-visible way, causing all future machines to have to live with a decision that made sense in the past. If CPUs are improved in cost/performance more rapidly than the I/O processor (and this will likely be the case), then moving the function may result in a slower machine in the next CPU.

The most telling example comes from the IBM 360. It was decided that the performance of the ISAM system, an early database system, would improve if some of the record searching occurred in the disk controller itself. A key field was associated with each record, and the device searched each key as the disk rotated until it found a match. It would then transfer the desired record. For the disk to find the key, there had to be an extra gap in the track. This scheme is applicable to searches through indices as well as data.

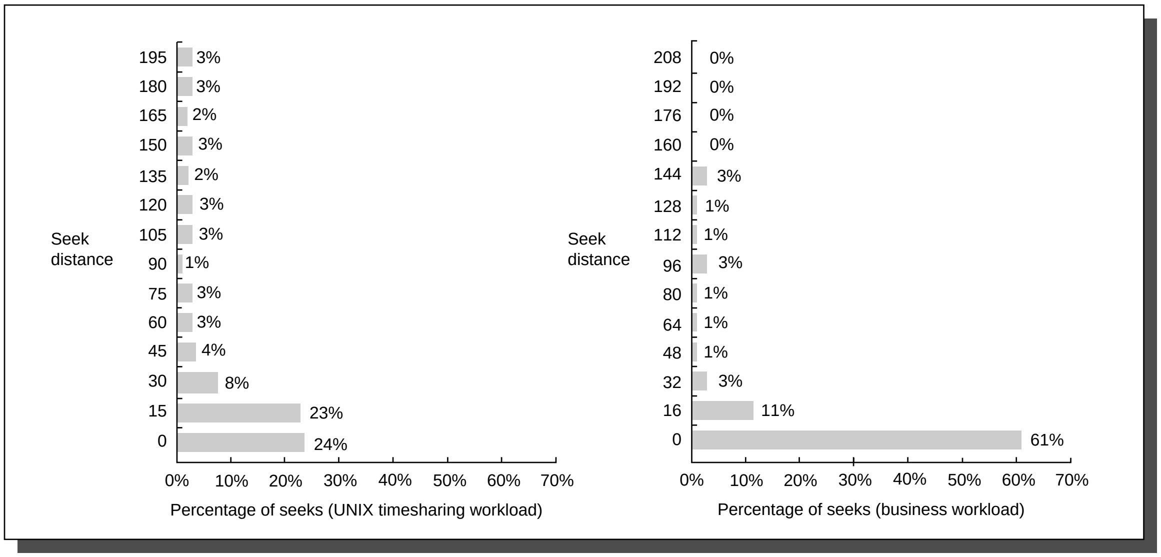


FIGURE 6.44 Sample measurements of seek distances for two systems. The measurements on the left were taken on a UNIX timesharing system. The measurements on the right were taken from a business-processing application in which the disk seek activity was scheduled. Seek distance of 0 means the access was made to the same cylinder. The rest of the numbers show the collective percentage for distances between numbers on the y axis. For example, 11% for the bar labeled 16 in the business graph means that the percentage of seeks between 1 and 16 cylinders was 11%. The UNIX measurements stopped at 200 cylinders, but this captured 85% of the accesses. The total was 1000 cylinders. The business measurements tracked all 816 cylinders of the disks. The only seek distances with 1% or greater of the seeks that are not in the graph are 224 with 4% and 304, 336, 512, and 624 each having 1%. This total is 94%, with the difference being small but nonzero distances in other categories. Measurements courtesy of Dave Anderson of Imprimis.

The speed at which a track can be searched is limited by the speed of the disk and of the number of keys that can be packed on a track. On an IBM 3330 disk, the key is typically 10 characters, but the total gap between records is equivalent to 191 characters if there were a key. (The gap is only 135 characters if there is no key, since there is no need for an extra gap for the key.) If we assume that the data is also 10 characters and that the track has nothing else on it, then a 13,165-byte track can contain

$$\frac{13{,}165}{191 + 10 + 10} = 62 \text{ key-data records}$$

This performance is

$$\frac{16.7 \text{ ms (1 revolution)}}{62} \approx .25 \text{ ms/key search}$$

In place of this scheme, we could put several key-data pairs in a single block and have smaller interrecord gaps. Assuming there are 15 key-data pairs per block and the track has nothing else on it, then

$$\frac{13{,}165}{135 + 15 \times (10 + 10)} = \frac{13{,}165}{135 + 300} = 30 \text{ blocks of key-data pairs}$$

The revised performance is then

$$\frac{16.7 \text{ ms (1 revolution)}}{30 \times 15} \approx 0.04 \text{ ms/key search}$$

Yet as CPUs got faster, the CPU time for a search was trivial. Although the strategy made early machines faster, programs that use the search-key operation in the I/O processor run almost six times slower on today's machines!

6.10 Concluding Remarks

According to Amdahl's Law, ignorance of I/O will lead to wasted performance as CPUs get faster. Disk performance is growing at 4% to 6% per year, while CPU performance is growing at a much faster rate. This performance gap has led to novel organizations to try to bridge it: file caches to improve latency and RAIDs to improve throughput. The future demands for I/O include better algorithms, better organizations, and more caching in a struggle to keep pace.

Nevertheless, the impressive improvement in capacity and cost per megabyte of disks and tape have made digital libraries plausible, whereby all of humankind's knowledge could be at the beck and call of your fingertips. Getting those requests to the libraries and the information back is the challenge of interconnection networks, the topic of the next chapter.

6.11 Historical Perspective and References

Mass storage is a term used there to imply a unit capacity in excess of one million alphanumeric characters…

Hoagland [1963]

Magnetic recording was invented to record sound, and by 1941 magnetic tape was able to compete with other storage devices. It was the success of the ENIAC in 1947 that led to the push to use tapes to record digital information. Reels of magnetic tapes dominated removable storage through the 1970s. In the 1980s the IBM 3480 cartridge became the de facto standard, at least for mainframes. It can transfer at 3 MB/sec since it reads 18 tracks in parallel. The capacity is just 200

MB for this 1/2-inch tape. In 1995 3M and IBM announced the IBM 3590, which transfers at 9 MB/sec and stores 10,000 MB. This device records the tracks in a zig-zag fashion rather than just longitudinally, so that the head reverses direction to follow the track. Its official name is *serpentine recording*. The other competitor is helical scan, which rotates the head to get the increased recording density. In 1995 the 8-mm tapes contain 6000 MB and transfer at about 1 MB/sec.Whatever their density and cost, the serial nature of tapes creates an appetite for storage devices with random access.

The magnetic disk first appeared in 1956 in the IBM Random Access Method of Accounting and Control (RAMAC) machine. This disk used 50 platters that were 24 inches in diameter, with a total capacity of 5 MB and an access time of 1 second. IBM maintained its leadership in the disk industry, and many of the future leaders of competing disk industries started their careers at IBM. The disk industry is responsible for 90% of the mass storage market.

Although RAMAC contained the first disk, the breakthrough in magnetic recording was found in later disks with air-bearing read-write heads. These allowed the head to ride on a cushion of air created by the fast-moving disk surface. This cushion meant the head could both follow imperfections in the surface and yet be very close to the surface. In 1995 heads fly 4 microinches above the surface, whereas the RAMAC drive was 1000 microinches away. Subsequent advances have been largely from improved quality of components and higher precision.

The second breakthough was the so-called Winchester disk design in about 1965. Before this time the cost of the electronics to control the disk meant that the media had to be removable. The integrated circuit lowered the costs of not only CPUs, but also of disk controllers and the electronics to control the arms. This price reduction meant that the media could be sealed with the reader. The sealed system meant the heads could fly closer to the surface, which led to increases in areal density. The IBM 1311 disk in 1962 had an areal density of 50,000 bits per square inch and a cost of about $800 per megabyte, and in 1995 IBM sells a disk using 640 million bits per square inch with a street price of about $0.25 per megabyte. (See Hospodor and Hoagland [1993] for more on magnetic storage trends.)

The personal computer created a market for small form-factor disk drives, since the 14-inch disk drives used in mainframes were bigger than the PC. In 1995 the 3.5-inch drive is the market leader, although the smaller 2.5-inch drive needed for portable computers is catching up quickly in sales volume. It remains to be seen whether hand-held devices, requiring even smaller disks, will become as popular as PCs or portables. These smaller disks inspired RAID; Chen et al. [1994] survey the RAID ideas and future directions.

One attraction of a personal computer is that you don't have to share it with anyone. This means that response time is predictable, unlike timesharing systems. Early experiments in the importance of fast response time were performed by Doherty and Kelisky [1979]. They showed that if computer-system response time increased one second, then user think time did also. Thadhani [1981] showed a jump in productivity as computer response times dropped to one second and another jump as they dropped to one-half second. His results inspired a flock of studies, and they supported his observations [IBM 1982]. In fact, some studies were started to disprove his results! Brady [1986] proposed differentiating entry time from think time (since entry time was becoming significant when the two were lumped together) and provided a cognitive model to explain the more-than-linear relationship between computer response time and user think time.

The ubiquitous microprocessor has inspired not only personal computers in the 1970s, but also the current trend to moving controller functions into I/O devices in the late 1980s and 1990s. I/O devices continued this trend by moving controllers into the devices themselves. These are called *intelligent devices*, and some bus standards (e.g., IPI and SCSI) have been created specifically for them. Intelligent devices can relax the timing constraints by handling many of the low-level tasks and queuing the results. For example, many SCSI-compatible disk drives include a track buffer on the disk itself, supporting read ahead and connect/disconnect. Thus, on a SCSI string some disks can be seeking and others loading their track buffer while one is transferring data from its buffer over the SCSI bus. The controller in the original RAMAC, built from vacuum tubes, only needed to move the head over the desired track, wait for the data to pass under the head, and transfer data with calculated parity.

SCSI, which stands for *small computer systems interface*, is an example of one company inventing a bus and generously encouraging other companies to build devices that would plug into it. This bus, originally called SASI, was invented by Shugart and was later standardized by the IEEE. Perhaps the first multivendor bus was the PDP-11 Unibus in 1970 from DEC. Alas, this open-door policy on buses is in contrast to companies with proprietary buses using patented interfaces, thereby preventing competition from plug-compatible vendors. This practice also raises costs and lowers availability of I/O devices that plug into proprietary buses, since such devices must have an interface designed just for that bus. The PCI bus being pushed by Intel gives us hope in 1995 of a return to open, standard I/O buses inside computers. There are also several candidates to be the successor to SCSI, most using simpler connectors and serial cables.

The machines of the RAMAC era gave us I/O interrupts as well as storage devices. The first machine to extend interrupts from detecting arithmetic abnormalities to detecting asynchronous I/O events is credited as the NBS DYSEAC in 1954 [Leiner and Alexander 1954]. The following year, the first machine with DMA was operational, the IBM SAGE. Just as today's DMA has, the SAGE had address counters that performed block transfers in parallel with CPU operations. (Smotherman [1989] explores the history of I/O in more depth.)

References

ANON, ET AL. [1985]. "A measure of transaction processing power," Tandem Tech. Rep. TR 85.2. Also appeared in *Datamation*, April 1, 1985.

BAKER, M. G., J. H. HARTMAN, M. D. KUPFER, K. W. SHIRRIFF, AND J. K. OUSTERHOUT [1991]. "Measurements of a distributed file system," *Proc. 13th ACM Symposium on Operating Systems*

Principles (October), 198–212.

BASHE, C. J., W. BUCHHOLZ, G. V. HAWKINS, J. L. INGRAM, AND N. ROCHESTER [1981]. “The architecture of IBM’s early computers,” *IBM J. Research and Development* 25:5 (September), 363–375.

BASHE, C. J., L. R. JOHNSON, J. H. PALMER, AND E. W. PUGH [1986]. *IBM’s Early Computers,* MIT Press, Cambridge, Mass.

BRADY, J. T. [1986]. “A theory of productivity in the creative process,” *IEEE CG&A* (May), 25–34.

BUCHER, I. V. AND A. H. HAYES [1980]. “I/O performance measurement on Cray-1 and CDC 7000 computers,” *Proc. Computer Performance Evaluation Users Group, 16th Meeting,* NBS 500-65, 245–254.

CHEN, P. M. AND D. A. PATTERSON [1993]. “Storage performance-metrics and benchmarks.” *Proc. IEEE* 81:8 (August), 1151–65.

CHEN, P. M. AND D. A. PATTERSON [1994a]. “Unix I/O performance in workstations and mainframes,” Tech. Rep. CSE-TR-200-94, Univ. of Michigan (March).

CHEN, P. M. AND D. A. PATTERSON [1994b]. “A new approach to I/O performance evaluation—Self-scaling I/O benchmarks, predicted I/O performance,” *ACM Trans. on Computer Systems* 12:4 (November).

CHEN, P. M., G. A. GIBSON, R. H. KATZ, AND D. A. PATTERSON [1990]. “An evaluation of redundant arrays of inexpensive disks using an Amdahl 5890,” *Proc. 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (May), Boulder, Colo.

CHEN, P. M., E. K. LEE, G. A. GIBSON, R. H. KATZ, AND D. A. PATTERSON [1994]. “RAID: High-performance, reliable secondary storage,” *ACM Computing Surveys* 26:2 (June), 145–88.

CHEN, P. M. AND E. K. LEE [1995]. “Striping in a RAID level 5 disk array,” *Proc. 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (May), 136–145.

DOHERTY, W. J. AND R. P. KELISKY [1979]. “Managing VM/CMS systems for user effectiveness,” *IBM Systems J.* 18:1, 143–166.

FEIERBACK, G. AND D. STEVENSON [1979]. “The Illiac-IV,” in *Infotech State of the Art Report on Supercomputers,* Maidenhead, England. This data also appears in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples* (1982), McGraw-Hill, New York, 268–269.

FRIESENBORG, S. E. AND R. J. WICKS [1985]. “DASD expectations: The 3380, 3380-23, and MVS/XA,” Tech. Bulletin GG22-9363-02 (July 10), Washington Systems Center.

GOLDSTEIN, S. [1987]. “Storage performance—An eight year outlook,” Tech. Rep. TR 03.308-1 (October), Santa Teresa Laboratory, IBM, San Jose, Calif.

GRAY, J. (ED.) [1993]. *The Benchmark Handbook for Database and Transaction Processing Systems,* 2nd ed. Morgan Kaufmann Publishers, San Francisco.

GRAY, J. AND A. REUTER [1993]. *Transaction Processing: Concepts and Techniques,* Morgan Kaufmann Publishers, San Francisco.

HARTMAN J. H. AND J. K. OUSTERHOUT [1993]. “Letter to the editor,” *ACM SIGOPS Operating Systems Review* 27:1 (January), 7–10.

HENLY, M. AND B. MCNUTT [1989]. “DASD I/O characteristics: A comparison of MVS to VM,” Tech. Rep. TR 02.1550 (May), IBM, General Products Division, San Jose, Calif.

HOAGLAND, A. S. [1963]. *Digital Magnetic Recording*, Wiley, New York.

HOSPODOR, A. D. AND A. S. HOAGLAND [1993]. “The changing nature of disk controllers.” *Proc. IEEE* 81:4 (April), 586–94.

HOWARD, J. H., ET AL. [1988]. “Scale and performance in a distributed file system,” *ACM Trans. on Computer Systems* 6:1, 51–81.

IBM [1982]. *The Economic Value of Rapid Response Time*, GE20-0752-0, White Plains, N.Y., 11–82.

Imprimis [1989]. *Imprimis Product Specification, 97209 Sabre Disk Drive IPI-2 Interface 1.2 GB,* Document No. 64402302 (May).

Jain, R. [1991]. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling,* Wiley, New York.

Kahn, R. E. [1972]. "Resource-sharing computer communication networks," *Proc. IEEE* 60:11 (November), 1397-1407.

Katz, R. H., D. A. Patterson, and G. A. Gibson [1990]. "Disk system architectures for high performance computing," *Proc. IEEE* 78:2 (February).

Kim, M. Y. [1986]. "Synchronized disk interleaving," *IEEE Trans. on Computers* C-35:11 (November).

Leiner, A. L. [1954]. "System specifications for the DYSEAC," *J. ACM* 1:2 (April), 57–81.

Leiner, A. L. and S. N. Alexander [1954]. "System organization of the DYSEAC," *IRE Trans. of Electronic Computers* EC-3:1 (March), 1–10.

Maberly, N. C. [1966]. *Mastering Speed Reading,* New American Library, New York.

Major, J. B. [1989]. "Are queuing models within the grasp of the unwashed?," *Proc. Int'l Conference on Management and Performance Evaluation of Computer Systems,* Reno, Nev. (December 11-15), 831–839.

Ousterhout, J. K., et al. [1985]. "A trace-driven analysis of the UNIX 4.2 BSD file system," *Proc. Tenth ACM Symposium on Operating Systems Principles,* Orcas Island, Wash., 15–24.

Patterson, D. A., G. A. Gibson, and R. H. Katz [1987]. "A case for redundant arrays of inexpensive disks (RAID)," Tech. Rep. UCB/CSD 87/391, Univ. of Calif. Also appeared in *ACM SIGMOD Conf. Proc.*, Chicago, June 1–3, 1988, 109–116.

Robinson, B. and L. Blount [1986]. "The VM/HPO 3880-23 performance results," IBM Tech. Bulletin GG66-0247-00 (April), Washington Systems Center, Gaithersburg, Md.

Salem, K. and H. Garcia-Molina [1986]. "Disk striping," *IEEE 1986 Int'l Conf. on Data Engineering*.

Scranton, R. A., D. A. Thompson, and D. W. Hunter [1983]. "The access time myth," Tech. Rep. RC 10197 (45223) (September 21), IBM, Yorktown Heights, N.Y.

Smith, A. J. [1985]. "Disk cache—Miss ratio analysis and design considerations," *ACM Trans. on Computer Systems* 3:3 (August), 161–203.

Smotherman, M. [1989]. "A sequencing-based taxonomy of I/O systems and review of historical machines," *Computer Architecture News* 17:5 (September), 5–15.

Thadhani, A. J. [1981]. "Interactive user productivity," *IBM Systems J.* 20:4, 407–423.

Thisquen, J. [1988]. "Seek time measurements," Amdahl Peripheral Products Division Tech. Rep. (May).

E X E R C I S E S

6.1 [10] <6.9> Using the formulas in the fallacy starting on page 549, including the caption of Figure 6.43 (page 551), calculate the seek time for moving the arm over one-third of the cylinders of the disk in Figure 6.2 (page 490).

6.2 [25] <6.9> Using the formulas in the fallacy starting on page 549, including the caption of Figure 6.43 (page 551), write a short program to calculate the "average" seek time by estimating the time for all possible seeks using these formulas and then dividing by the number of seeks. How close is the answer to Exercise 6.1 to this answer?

6.3 [20] <6.9> Using the formulas in the fallacy starting on page 549, including the caption of Figure 6.43 (page 551) and the statistics in Figure 6.44 (page 552), calculate the average seek distance on the disk in Figure 6.2 (page 490). Use the midpoint of a range as the seek distance. For example, use 98 as the seek distance for the entry representing 91–105 in Figure 6.44. For the business workload, just ignore the missing 5% of the seeks. For the UNIX workload, assume the missing 15% of the seeks have an average distance of 300 cylinders. If you were misled by the fallacy, you might calculate the average distance as 884/3. What is the measured distance for each workload?

6.4 [20] <6.9> Figure 6.2 (page 490) gives the manufacturer's average seek time. Using the formulas in the fallacy starting on page 549, including the equations in Figure 6.43 (page 551), and the statistics in Figure 6.44 (page 552), what is the average seek time for each workload on the disk in Figure 6.2 using the measurements? Make the same assumptions as in Exercise 6.3.

6.5 [20/15/15/15/15/15] <6.4> The I/O bus and memory system of a computer are capable of sustaining 1000 MB/sec without interfering with the performance of an 800-MIPS CPU (costing $50,000). Here are the assumptions about the software:

- Each transaction requires 2 disk reads plus 2 disk writes.
- The operating system uses 15,000 instructions for each disk read or write.
- The database software executes 40,000 instructions to process a transaction.
- The transfer size is 100 bytes.

You have a choice of two different types of disks:

- A small disk that stores 500 MB and costs $100.
- A big disk that stores 1250 MB and costs $250.

Either disk in the system can support on average 30 disk reads or writes per second.

Answer parts (a)–(f) using the TPS benchmark in section 6.4. Assume that the requests are spread evenly to all the disks, that there is no waiting time due to busy disks, and that the account file must be large enough to handle 1000 TPS according to the benchmark ground rules.

a. [20] <6.4> How many TPS transactions per second are possible with each disk organization, assuming that each uses the minimum number of disks to hold the account file?

b. [15] <6.4> What is the system cost per transaction per second of each alternative for TPS?

c. [15] <6.4> How fast does a CPU need to be to make the 1000 MB/sec I/O bus a bottleneck for TPS? (Assume that you can continue to add disks.)

d. [15] <6.4> As manager of MTP (Mega TP), you are deciding whether to spend your development money building a faster CPU or improving the performance of the software. The database group says they can reduce a transaction to 1 disk read and 1 disk write and cut the database instructions per transaction to 30,000. The hardware group can build a faster CPU that sells for the same amount as the slower CPU with the same development budget. (Assume you can add as many disks as needed to get higher performance.) How much faster does the CPU have to be to match the performance gain of the software improvement?

e. [15] <6.4> The MTP I/O group was listening at the door during the software presentation. They argue that advancing technology will allow CPUs to get faster without significant investment, but that the cost of the system will be dominated by disks if they don't develop new small, faster disks. Assume the next CPU is 100% faster at the same cost and that the new disks have the same capacity as the old ones. Given the new CPU and the old software, what will be the cost of a system with enough old small disks so that they do not limit the TPS of the system?

f. [15] <6.4> Start with the same assumptions as in part (e). Now assume that you have as many new disks as you had old small disks in the original design. How fast must the new disks be (I/Os per second) to achieve the same TPS rate with the new CPU as the system in part (e)? What will the system cost?

6.6 [20] <6.4> Assume that we have the following two magnetic-disk configurations: a single disk and an array of four disks. Each disk has 20 surfaces, 885 tracks per surface, and 16 sectors/track. Each sector holds 1K bytes, and it revolves at 7200 RPM. Use the seek-time formula in the fallacy starting on page 549, including the equations in Figure 6.43 (page 551). The time to switch between surfaces is the same as to move the arm one track. In the disk array all the spindles are synchronized—sector 0 in every disk rotates under the head at the exact same time—and the arms on all four disks are always over the same track. The data is "striped" across all four disks, so four consecutive sectors on a single-disk system will be spread one sector per disk in the array. The delay of the disk controller is 2 ms per transaction, either for a single disk or for the array. Assume the performance of the I/O system is limited only by the disks and that there is a path to each disk in the array. Calculate the performance in both I/Os per second and megabytes per second of these two disk organizations, assuming the request pattern is random reads of 4 KB of sequential sectors. Assume the 4 KB are aligned under the same arm on each disk in the array.

6.7 [20]<6.4> Start with the same assumptions as in Exercise 6.5 (e). Now calculate the performance in both I/Os per second and megabytes per second of these two disk organizations assuming the request pattern is reads of 4 KB of sequential sectors where the average seek distance is 10 tracks. Assume the 4 KB are aligned under the same arm on each disk in the array.

6.8 [20] <6.4> Start with the same assumptions as in Exercise 6.5 (e). Now calculate the performance in both I/Os per second and megabytes per second of these two disk organizations assuming the request pattern is random reads of 1 MB of sequential sectors. (If it matters, assume the disk controller allows the sectors to arrive in any order.)

6.9 [20] <6.2> Assume that we have one disk defined as in Exercise 6.5 (e). Assume that we read the next sector after any read and that *all* read requests are one sector in length. We store the extra sectors that were read ahead in a disk cache. Assume that the probability of receiving a request for the sector we read ahead at some time in the future (before it must be discarded because the disk-cache buffer fills) is 0.1. Assume that we must still pay the controller overhead on a disk-cache read hit, and the transfer time for the disk cache is 250 ns per word. Is the read-ahead strategy faster? (*Hint*: Solve the problem in the steady state by assuming that the disk cache contains the appropriate information and a request has just missed.)

6.10 [20/10/20/20] <6.4–6.6> Assume the following information about our DLX machine:

- Loads 2 cycles.
- Stores 2 cycles.
- All other instructions are 1 cycle.

Use the summary instruction mix information on DLX for gcc from Chapter 2.

Here are the cache statistics for a write-through cache:

- Each cache block is four words, and the whole block is read on any miss.
- Cache miss takes 23 cycles.
- Write through takes 16 cycles to complete, and there is no write buffer.

Here are the cache statistics for a write-back cache:

- Each cache block is four words, and the whole block is read on any miss.
- Cache miss takes 23 cycles for a clean block and 31 cycles for a dirty block.
- Assume that on a miss, 30% of the time the block is dirty.

Assume that the bus

- Is only busy during transfers
- Transfers on average 1 word / clock cycle
- Must read or write a single word at a time (it is not faster to access two at once)

a. [20] <6.4–6.6> Assume that DMA I/O can take place simultaneously with CPU cache hits. Also assume that the operating system can guarantee that there will be no stale-data problem in the cache due to I/O. The sector size is 1 KB. Assume the cache miss rate is 5%. On the average, what percentage of the bus is used for each cache write policy? (This measured is called the *traffic ratio* in cache studies.)

b. [10] <6.4–6.6> Start with the same assumptions as in part (a). If the bus can be loaded up to 80% of capacity without suffering severe performance penalties, how much memory bandwidth is available for I/O for each cache write policy? The cache miss rate is still 5%.

c. [20] <6.4–6.6> Start with the same assumptions as in part (a). Assume that a disk sector read takes 1000 clock cycles to initiate a read, 100,000 clock cycles to find the data on the disk, and 1000 clock cycles for the DMA to transfer the data to memory. How many disk reads can occur per million instructions executed for each write policy? How does this change if the cache miss rate is cut in half?

d. [20] <6.4–6.6> Start with the same assumptions as in part (c). Now you can have any number of disks. Assuming ideal scheduling of disk accesses, what is the maximum number of sector reads that can occur per million instructions executed?

6.11 [50] < 6.4> Take your favorite computer and write a program that achieves maximum bandwidth to and from disks. What is the percentage of the bandwidth that you achieve compared with what the I/O device manufacturer claims?

6.12 [20] <6.2,6.5> Search the World Wide Web to find descriptions of recent magnetic disks of different diameters. Be sure to include at least the information in Figure 6.2 on page 490.

6.13 [20] <6.9> Using data collected in Exercise 6.12, plot the two projections of seek time as used in Figure 6.43 (page 551). What seek distance has the largest percentage of difference between these two predictions? If you have the real seek distance data from Exercise 6.12, add that data to the plot and see on average how close each projection is to the real seek times.

6.14 [15] <6.2,6.5> Using the answer to Exercise 6.13, which disk would be a good building block to build a 100-GB storage subsystem using mirroring (RAID 1)? Why?

6.15 [15] <6.2,6.5> Using the answer to Exercise 6.13, which disk would be a good building block to build a 1000-GB storage subsystem using distributed parity (RAID 5)? Why?

6.16 [15] <6.4> Starting with the Example on page 515, calculate the average length of the queue and the average length of the system.

6.17 [15] <6.4> Redo the Example that starts on page 515, but this time assume the distribution of disk service times has a squared coefficient of variance of 2.0 (C = 2.0), versus 1.0 in the Example. How does this change affect the answers?

6.18 [20] <6.7> The I/O utilization rules of thumb on page 535 are just guidelines and are subject to debate. Redo the Example starting on page 535, but increase the limit of SCSI utilization to 50%, 60%, ..., until it is never the bottleneck. How does this change affect the answers? What is the new bottleneck? (*Hint*: Use a spreadsheet program to find answers.)

6.19 [15]<6.2> Tape libraries were invented as archival storage, and hence have relatively few readers per tape. Calculate how long it would take to read all the data for a system with 6000 tapes, 10 readers that read at 9 MB/sec, and 30 seconds per tape to put the old tape away and load a new tape.

6.20 [25]<6.2>Extend the figures, showing price per system and price per megabyte of disks by collecting data from advertisements in the January issues of *Byte* magazine after 1995. How fast are prices changing now?