6

Storage Systems

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
a pioneer of magnetic disks
(1982)

Combining bandwidth and storage . . . enables swift and reliable access to the ever expanding troves of content on the proliferating disks and . . . repositories of the Internet. . . . the capacity of storage arrays of all kinds is rocketing ahead of the advance of computer performance.

George Gilder
"The End Is Drawing Nigh,"
Forbes ASAP (April 4, 2000)

6.1 Introduction

The popularity of Internet services like search engines and auctions has enhanced the importance of I/O for computers, since no one would want a desktop computer that couldn't access the Internet. This rise in importance of I/O is reflected by the names of our times. The 1960s to 1980s were called the Computing Revolution; the period since 1990 has been called the Information Age, with concerns focused on advances in information technology versus raw computational power. Internet services depend upon massive storage, which is the focus of this chapter, and networking, which is the focus of Appendix E.

This shift in focus from computation to communication and storage of information emphasizes reliability and scalability as well as cost-performance. Although it is frustrating when a program crashes, people become hysterical if they lose their data. Hence, storage systems are typically held to a higher standard of dependability than the rest of the computer. Dependability is the bedrock of storage, yet it also has its own rich performance theory—queuing theory—that balances throughput versus response time. The software that determines which processor features get used is the compiler, but the operating system usurps that role for storage.

Thus, storage has a different, multifaceted culture from processors, yet it is still found within the architecture tent. We start our exploration with advances in magnetic disks, as they are the dominant storage device today in desktop and server computers. We assume readers are already familiar with the basics of storage devices, some of which were covered in Chapter 1.

6.2 Advanced Topics in Disk Storage

The disk industry historically has concentrated on improving the capacity of disks. Improvement in capacity is customarily expressed as improvement in *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. Between then and about 1996, the rate improved to 60% per year, quadrupling density every three years and matching the traditional rate of DRAMs. From 1997 to about 2003, the rate increased to 100%, or doubling every year. After the innovations that allowed the renaissances had largely played out, the rate has dropped recently to about 30% per year. In 2006, the highest density in commercial products is 130 billion bits per square inch. Cost per gigabyte has dropped at least as fast as areal density has increased, with smaller diameter drives playing the larger role in this improvement. Costs per gigabyte improved by a factor of 100,000 between 1983 and 2006.

Magnetic disks have been challenged many times for supremacy of secondary storage. Figure 6.1 shows one reason: the fabled *access time gap* between disks and DRAM. DRAM latency is about 100,000 times less than disk, and that performance advantage costs 30–150 times more per gigabyte for DRAM.

The bandwidth gap is more complex. For example, a fast disk in 2006 transfers about 115 MB/sec from the disk media with 37 GB of storage and costs about $150 (as we will see later in Figure 6.3). A 2 GB DRAM module costing about $300 in 2006 could transfer at 3200 MB/sec (see Section 5.3 in Chapter 5), giving the DRAM module about 28 times higher bandwidth than the disk. However, the bandwidth per GB is 500 times higher for DRAM, and the bandwidth per dollar is 14 times higher.

Many have tried to invent a technology cheaper than DRAM but faster than disk to fill that gap, but thus far, all have failed. Challengers have never had a product to market at the right time. By the time a new product would ship, DRAMs and disks have made advances as predicted earlier, costs have dropped accordingly, and the challenging product is immediately obsolete.

The closest challenger is flash memory. This semiconductor memory is nonvolatile like disks, and it has about the same bandwidth as disks, but latency is 100–1000 times faster than disk. In 2006, the price per gigabyte of flash was about the same as DRAM. Flash is popular in cameras and portable music players because it comes in much smaller capacities and it is more power efficient than disks, despite the cost per gigabyte being 50 times higher than disks. Unlike

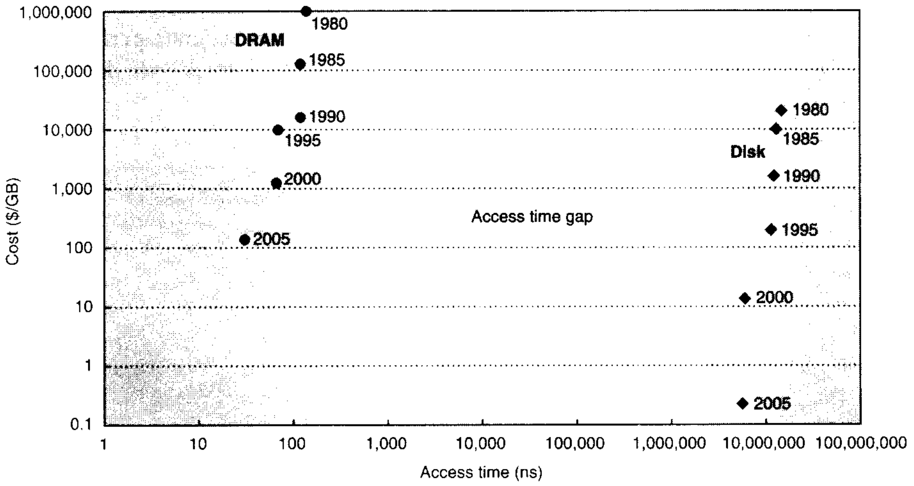


Figure 6.1 Cost versus access time for DRAM and magnetic disk in 1980, 1985, 1990, 1995, 2000, and 2005. The two-order-of-magnitude gap in cost and five-order-of-magnitude gap in access times between semiconductor memory and rotating magnetic disks has inspired a host of competing technologies to try to fill them. So far, such attempts have been made obsolete before production by improvements in magnetic disks, DRAMs, or both. Note that between 1990 and 2005 the cost per gigabyte DRAM chips made less improvement, while disk cost made dramatic improvement.

disks and DRAM, flash memory bits wear out—typically limited to 1 million writes—and so they are not popular in desktop and server computers.

While disks will remain viable for the foreseeable future, the conventional sector-track-cylinder model did not. The assumptions of the model are that nearby blocks are on the same track, blocks in the same cylinder take less time to access since there is no seek time, and some tracks are closer than others.

First, disks started offering higher-level intelligent interfaces, like ATA and SCSI, when they included a microprocessor inside a disk. To speed up sequential transfers, these higher-level interfaces organize disks more like tapes than like random access devices. The logical blocks are ordered in serpentine fashion across a single surface, trying to capture all the sectors that are recorded at the same bit density. (Disks vary the recording density since it is hard for the electronics to keep up with the blocks spinning much faster on the outer tracks, and lowering linear density simplifies the task.) Hence, sequential blocks may be on different tracks. We will see later in Figure 6.22 on page 401 an illustration of the fallacy of assuming the conventional sector-track model when working with modern disks.

Second, shortly after the microprocessors appeared inside disks, the disks included buffers to hold the data until the computer was ready to accept it, and later caches to avoid read accesses. They were joined by a command queue that allowed the disk to decide in what order to perform the commands to maximize performance while maintaining correct behavior. Figure 6.2 shows how a queue depth of 50 can double the number of I/Os per second of random I/Os due to better scheduling of accesses. Although it's unlikely that a system would really have 256 commands in a queue, it would triple the number of I/Os per second. Given buffers, caches, and out-of-order accesses, an accurate performance model of a real disk is much more complicated than sector-track-cylinder.

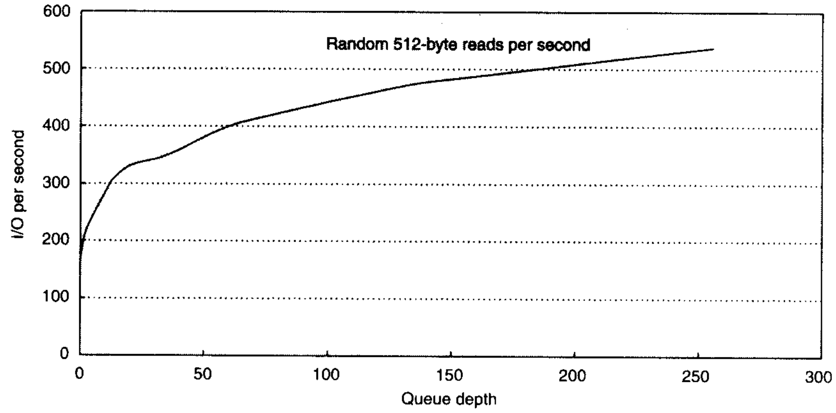


Figure 6.2 Throughput versus command queue depth using random 512-byte reads. The disk performs 170 reads per second starting at no command queue, and doubles performance at 50 and triples at 256 [Anderson 2003].

Finally, the number of platters shrank from 12 in the past to 4 or even 1 today, so the cylinder has less importance than before since the percentage of data in a cylinder is much less.

Disk Power

Power is an increasing concern for disks as well as for processors. A typical ATA disk in 2006 might use 9 watts when idle, 11 watts when reading or writing, and 13 watts when seeking. Because it is more efficient to spin smaller mass, smaller-diameter disks can save power. One formula that indicates the importance of rotation speed and the size of the platters for the power consumed by the disk motor is the following [Gurumurthi 2005]:

$$\text{Power} \approx \text{Diameter}^{4.6} \times \text{RPM}^{2.8} \times \text{Number of platters}$$

Thus, smaller platters, slower rotation, and fewer platters all help reduce disk motor power, and most of the power is in the motor.

Figure 6.3 shows the specifications of two 3.5-inch disks in 2006. The *Serial ATA* (SATA) disks shoot for high capacity and the best cost per gigabyte, and so the 500 GB drives cost less than $1 per gigabyte. They use the widest platters that fit the form factor and use four or five of them, but they spin at 7200 RPM and seek relatively slowly to lower power. The corresponding *Serial Attach SCSI* (SAS) drive aims at performance, and so it spins at 15,000 RPM and seeks much faster. To reduce power, the platter is much narrower than the form factor and it has only a single platter. This combination reduces capacity of the SAS drive to 37 GB.

The cost per gigabyte is about a factor of five better for the SATA drives, and conversely, the cost per I/O per second or MB transferred per second is about a factor of five better for the SAS drives. Despite using smaller platters and many fewer of them, the SAS disks use twice the power of the SATA drives, due to the much faster RPM and seeks.

	Capacity (GB)	**Price**	**Platters**	**RPM**	**Diameter (inches)**	**Average seek (ms)**	**Power (watts)**	**I/O/sec**	**Disk BW (MB/sec)**	**Buffer BW (MB/sec)**	**Buffer size (MB)**	**MTTF (hrs)**
SATA	500	$375	4 or 5	7,200	3.7	8–9	12	117	31–65	300	16	0.6M
SAS	37	$150	1	15,000	2.6	3–4	25	285	85–142	300	8	1.2M

Figure 6.3 Serial ATA (SATA) versus Serial Attach SCSI (SAS) drives in 3.5-inch form factor in 2006. The I/Os per second are calculated using the average seek plus the time for one-half rotation plus the time to transfer one sector of 512 KB.

Advanced Topics in Disk Arrays

An innovation that improves both dependability and performance of storage systems is *disk arrays*. One argument for arrays is that potential throughput can be increased by having many disk drives and, hence, many disk arms, rather than fewer large drives. Simply spreading data over multiple disks, called *striping*, automatically forces accesses to several disks if the data files are large. (Although arrays improve throughput, latency is not necessarily improved.) As we saw in Chapter 1, the drawback is that with more devices, dependability decreases: N devices generally have $1/N$ the reliability of a single device.

Although a disk array would have more faults than a smaller number of larger disks when each disk has the same reliability, dependability is improved by adding redundant disks to the array to tolerate faults. That is, if a single disk fails, the lost information is reconstructed from redundant information. The only danger is in having another disk fail during the *mean time to repair* (MTTR). Since the *mean time to failure* (MTTF) of disks is tens of years, and the MTTR is measured in hours, redundancy can make the measured reliability of many disks much higher than that of a single disk.

Such redundant disk arrays have become known by the acronym *RAID*, standing originally for *redundant array of inexpensive disks*, although some prefer the word *independent* for *I* in the acronym. The ability to recover from failures plus the higher throughput, either measured as megabytes per second or as I/Os per second, makes RAID attractive. When combined with the advantages of smaller size and lower power of small-diameter drives, RAIDs now dominate large-scale storage systems.

Figure 6.4 summarizes the five standard RAID levels, showing how eight disks of user data must be supplemented by redundant or check disks at each RAID level, and lists the pros and cons of each level. The standard RAID levels are well documented, so we will just do a quick review here and discuss advanced levels in more depth.

- *RAID 0*—It has no redundancy and is sometimes nicknamed *JBOD*, for "just a bunch of disks," although the data may be striped across the disks in the array. This level is generally included to act as a measuring stick for the other RAID levels in terms of cost, performance, and dependability.
- *RAID 1*—Also called *mirroring* or *shadowing*, there are two copies of every piece of data. It is the simplest and oldest disk redundancy scheme, but it also has the highest cost. Some array controllers will optimize read performance by allowing the mirrored disks to act independently for reads, but this optimization means it may take longer for the mirrored writes to complete.
- *RAID 2*—This organization was inspired by applying memory-style error correcting codes to disks. It was included because there was such a disk array product at the time of the original RAID paper, but none since then as other RAID organizations are more attractive.

RAID level		Disk failures tolerated, check space overhead for 8 data disks	Pros	Cons	Company products
0	Nonredundant striped	0 failures, 0 check disks	No space overhead	No protection	Widely used
1	Mirrored	1 failure, 8 check disks	No parity calculation; fast recovery; small writes faster than higher RAIDs; fast reads	Highest check storage overhead	EMC, HP (Tandem), IBM
2	Memory-style ECC	1 failure, 4 check disks	Doesn't rely on failed disk to self-diagnose	~ Log 2 check storage overhead	Not used
3	Bit-interleaved parity	1 failure, 1 check disk	Low check overhead; high bandwidth for large reads or writes	No support for small, random reads or writes	Storage Concepts
4	Block-interleaved parity	1 failure, 1 check disk	Low check overhead; more bandwidth for small reads	Parity disk is small write bottleneck	Network Appliance
5	Block-interleaved distributed parity	1 failure, 1 check disk	Low check overhead; more bandwidth for small reads and writes	Small writes → 4 disk accesses	Widely used
6	Row-diagonal parity, EVEN-ODD	2 failures, 2 check disks	Protects against 2 disk failures	Small writes → 6 disk accesses; 2X check overhead	Network Appliance

Figure 6.4 RAID levels, their fault tolerance, and their overhead in redundant disks. The paper that introduced the term *RAID* [Patterson, Gibson, and Katz 1987] used a numerical classification that has become popular. In fact, the nonredundant disk array is often called RAID 0, indicating the data are striped across several disks but without redundancy. Note that mirroring (RAID 1) in this instance can survive up to eight disk failures provided only one disk of each mirrored pair fails; worst case is both disks in a mirrored pair. In 2006, there may be no commercial implementations of RAID 2; the rest are found in a wide range of products. RAID 0 + 1, 1 + 0, 01, 10, and 6 are discussed in the text.

- *RAID 3*—Since the higher-level disk interfaces understand the health of a disk, it's easy to figure out which disk failed. Designers realized that if one extra disk contains the parity of the information in the data disks, a single disk allows recovery from a disk failure. The data is organized in stripes, with *N* data blocks and one parity block. When a failure occurs, you just "subtract" the good data from the good blocks, and what remains is the missing data. (This works whether the failed disk is a data disk or the parity disk.) RAID 3 assumes the data is spread across all disks on reads and writes, which is attractive when reading or writing large amounts of data.
- *RAID 4*—Many applications are dominated by small accesses. Since sectors have their own error checking, you can safely increase the number of reads per second by allowing each disk to perform independent reads. It would seem that writes would still be slow, if you have to read every disk to calculate parity. To increase the number of writes per second, an alternative

approach involves only two disks. First, the array reads the old data that is about to be overwritten, and then calculates what bits would change before it writes the new data. It then reads the old value of the parity on the check disks, updates parity according to the list of changes, and then writes the new value of parity to the check disk. Hence, these so-called "small writes" are still slower than small reads—they involve four disks accesses—but they are faster than if you had to read all disks on every write. RAID 4 has the same low check disk overhead as RAID 3, and it can still do large reads and writes as fast as RAID 3 in addition to small reads and writes, but control is more complex.

- *RAID 5*—Note that a performance flaw for small writes in RAID 4 is that they all must read and write the same check disk, so it is a performance bottleneck. RAID 5 simply distributes the parity information across all disks in the array, thereby removing the bottleneck. The parity block in each stripe is rotated so that parity is spread evenly across all disks. The disk array controller must now calculate which disk has the parity for when it wants to write a given block, but that can be a simple calculation. RAID 5 has the same low check disk overhead as RAID 3 and 4, and it can do the large reads and writes of RAID 3 and the small reads of RAID 4, but it has higher small write bandwidth than RAID 4. Nevertheless, RAID 5 requires the most sophisticated controller of the classic RAID levels.

Having completed our quick review of the classic RAID levels, we can now look at two levels that have become popular since RAID was introduced.

RAID 10 versus 01 (or 1 + 0 versus RAID 0+1)

One topic not always described in the RAID literature involves how mirroring in RAID 1 interacts with striping. Suppose you had, say, four disks worth of data to store and eight physical disks to use. Would you create four pairs of disks—each organized as RAID 1—and then stripe data across the four RAID 1 pairs? Alternatively, would you create two sets of four disks—each organized as RAID 0—and then mirror writes to both RAID 0 sets? The RAID terminology has evolved to call the former RAID 1 + 0 or RAID 10 ("striped mirrors") and the latter RAID 0 + 1 or RAID 01 ("mirrored stripes").

RAID 6: Beyond a Single Disk Failure

The parity-based schemes of the RAID 1 to 5 protect against a single self-identifying failure. However, if an operator accidentally replaces the wrong disk during a failure, then the disk array will experience two failures, and data will be lost. Another concern with is that since disk bandwidth is growing more slowly than disk capacity, the MTTR of a disk in a RAID system is increasing, which in turn increases the chances of a second failure. For example, a 500 GB SATA disk could take about 3 hours to read sequentially assuming no interference. Given that the damaged RAID is likely to continue to serve data, reconstruction could be stretched considerably, thereby increasing MTTR. Besides increasing reconstruction time, another concern is that reading much more data during reconstruction means increasing the chance of an uncorrectable media failure, which would result in data loss. Other arguments for concern about simultaneous multiple failures are the increasing number of disks in arrays and the use of ATA disks, which are slower and larger than SCSI disks.

Hence, over the years, there has been growing interest in protecting against more than one failure. Network Appliance, for example, started by building RAID 4 file servers. As double failures were becoming a danger to customers, they created a more robust scheme to protect data, called *row-diagonal parity* or *RAID-DP* [Corbett 2004]. Like the standard RAID schemes, row-diagonal parity uses redundant space based on a parity calculation on a per-stripe basis. Since it is protecting against a double failure, it adds two check blocks per stripe of data. Let's assume there are $p + 1$ disks total, and so $p - 1$ disks have data. Figure 6.5 shows the case when p is 5.

The row parity disk is just like in RAID 4; it contains the even parity across the other four data blocks in its stripe. Each block of the diagonal parity disk contains the even parity of the blocks in the same diagonal. Note that each diagonal does not cover one disk; for example, diagonal 0 does not cover disk 1. Hence, we need just $p - 1$ diagonals to protect the p disks, so the disk only has diagonals 0 to 3 in Figure 6.5.

Let's see how row-diagonal parity works by assuming that data disks 1 and 3 fail in Figure 6.5. We can't perform the standard RAID recovery using the first row using row parity, since it is missing two data blocks from disks 1 and 3. However, we can perform recovery on diagonal 0, since it is only missing the data block associated with disk 3. Thus, row-diagonal parity starts by recovering one of the four blocks on the failed disk in this example using diagonal parity. Since each diagonal misses one disk, and all diagonals miss a different disk, two diagonals are only missing one block. They are diagonals 0 and 2 in this example,

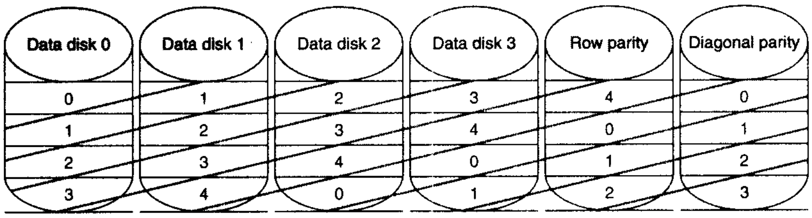


Figure 6.5 Row diagonal parity for $p = 5$, which protects four data disks from double failures [Corbett 2004]. This figure shows the diagonal groups for which parity is calculated and stored in the diagonal parity disk. Although this shows all the check data in separate disks for row parity and diagonal parity as in RAID 4, there is a rotated version of row-diagonal parity that is analogous to RAID 5. Parameter p must be prime and greater than 2. However, you can make p larger than the number of data disks by assuming the missing disks have all zeros, and the scheme still works. This trick makes it easy to add disks to an existing system. NetApp picks p to be 257, which allows the system to grow to up to 256 data disks.

so we next restore the block from diagonal 2 from failed disk 1. Once the data for those blocks is recovered, then the standard RAID recovery scheme can be used to recover two more blocks in the standard RAID 4 stripes 0 and 2, which in turn allows us to recover more diagonals. This process continues until two failed disks are completely restored.

The EVEN-ODD scheme developed earlier by researchers at IBM is similar to row diagonal parity, but it has a bit more computation during operation and recovery [Blaum 1995]. Papers that are more recent show how to expand EVEN-ODD to protect against three failures [Blaum 1996; Blaum 2001].

6.3 Definition and Examples of Real Faults and Failures

Although people may be willing to live with a computer that occasionally crashes and forces all programs to be restarted, they insist that their information is never lost. The prime directive for storage is then to remember information, no matter what happens.

Chapter 1 covered the basics of dependability, and this section expands that information to give the standard definitions and examples of failures.

The first step is to clarify confusion over terms. The terms *fault, error,* and *failure* are often used interchangeably, but they have different meanings in the dependability literature. For example, is a programming mistake a fault, error, or failure? Does it matter whether we are talking about when it was designed, or when the program is run? If the running program doesn't exercise the mistake, is it still a fault/error/failure? Try another one. Suppose an alpha particle hits a DRAM memory cell. Is it a fault/error/failure if it doesn't change the value? Is it a fault/error/failure if the memory doesn't access the changed bit? Did a fault/error/failure still occur if the memory had error correction and delivered the corrected value to the CPU? You get the drift of the difficulties. Clearly, we need precise definitions to discuss such events intelligently.

To avoid such imprecision, this subsection is based on the terminology used by Laprie [1985] and Gray and Siewiorek [1991], endorsed by IFIP working group 10.4 and the IEEE Computer Society Technical Committee on Fault Tolerance. We talk about a system as a single module, but the terminology applies to submodules recursively. Let's start with a definition of *dependability*:

> Computer system *dependability* is the quality of delivered service such that reliance can justifiably be placed on this service. The *service* delivered by a system is its observed *actual behavior* as perceived by other system(s) interacting with this system's users. Each module also has an ideal *specified behavior,* where a *service specification* is an agreed description of the expected behavior. A system *failure* occurs when the actual behavior deviates from the specified behavior. The failure occurred because of an *error,* a defect in that module. The cause of an error is a *fault.*
>
> When a fault occurs, it creates a *latent error,* which becomes *effective* when it is activated; when the error actually affects the delivered service, a failure occurs.

> The time between the occurrence of an error and the resulting failure is the *error latency*. Thus, an error is the manifestation *in the system* of a fault, and a failure is the manifestation *on the service* of an error. [p. 3]

Let's go back to our motivating examples above. A programming mistake is a *fault*. The consequence is an *error* (or *latent error*) in the software. Upon activation, the error becomes *effective*. When this effective error produces erroneous data that affect the delivered service, a *failure* occurs.

An alpha particle hitting a DRAM can be considered a fault. If it changes the memory, it creates an error. The error will remain latent until the affected memory word is read. If the effective word error affects the delivered service, a failure occurs. If ECC corrected the error, a failure would not occur.

A mistake by a human operator is a fault. The resulting altered data is an error. It is latent until activated, and so on as before.

To clarify, the relation between faults, errors, and failures is as follows:

- A fault creates one or more latent errors.
- The properties of errors are (1) a latent error becomes effective once activated; (2) an error may cycle between its latent and effective states; (3) an effective error often propagates from one component to another, thereby creating new errors. Thus, either an effective error is a formerly latent error in that component, or it has propagated from another error in that component or from elsewhere.
- A component failure occurs when the error affects the delivered service.
- These properties are recursive and apply to any component in the system.

Gray and Siewiorek classify faults into four categories according to their cause:

1. *Hardware faults*—Devices that fail, such as perhaps due to an alpha particle hitting a memory cell
2. *Design faults*—Faults in software (usually) and hardware design (occasionally)
3. *Operation faults*—Mistakes by operations and maintenance personnel
4. *Environmental faults*—Fire, flood, earthquake, power failure, and sabotage

Faults are also classified by their duration into transient, intermittent, and permanent [Nelson 1990]. *Transient faults* exist for a limited time and are not recurring. *Intermittent faults* cause a system to oscillate between faulty and fault-free operation. *Permanent faults* do not correct themselves with the passing of time.

Now that we have defined the difference between faults, errors, and failures, we are ready to see some real-world examples. Publications of real error rates are rare for two reasons. First, academics rarely have access to significant hardware resources to measure. Second, industrial researchers are rarely allowed to publish failure information for fear that it would be used against their companies in the marketplace. A few exceptions follow.

Berkeley's Tertiary Disk

The Tertiary Disk project at the University of California created an art image server for the Fine Arts Museums of San Francisco. This database consists of high-quality images of over 70,000 artworks. The database was stored on a cluster, which consisted of 20 PCs connected by a switched Ethernet and containing 368 disks. It occupied seven 7-foot-high racks.

Figure 6.6 shows the failure rates of the various components of Tertiary Disk. In advance of building the system, the designers assumed that SCSI data disks would be the least reliable part of the system, as they are both mechanical and plentiful. Next would be the IDE disks, since there were fewer of them, then the power supplies, followed by integrated circuits. They assumed that passive devices like cables would scarcely ever fail.

Figure 6.6 shatters some of those assumptions. Since the designers followed the manufacturer's advice of making sure the disk enclosures had reduced vibration and good cooling, the data disks were very reliable. In contrast, the PC chassis containing the IDE/ATA disks did not afford the same environmental controls. (The IDE/ATA disks did not store data, but helped the application and operating system to boot the PCs.) Figure 6.6 shows that the SCSI backplane, cables, and Ethernet cables were no more reliable than the data disks themselves!

As Tertiary Disk was a large system with many redundant components, it could survive this wide range of failures. Components were connected and mirrored images were placed so that no single failure could make any image unavailable. This strategy, which initially appeared to be overkill, proved to be vital.

This experience also demonstrated the difference between transient faults and hard faults. Virtually all the failures in Figure 6.6 appeared first as transient faults. It was up to the operator to decide if the behavior was so poor that they needed to be replaced or if they could continue. In fact, the word "failure" was not used; instead, the group borrowed terms normally used for dealing with problem employees, with the operator deciding whether a problem component should or should not be "fired."

Tandem

The next example comes from industry. Gray [1990] collected data on faults for Tandem Computers, which was one of the pioneering companies in fault-tolerant computing and used primarily for databases. Figure 6.7 graphs the faults that caused system failures between 1985 and 1989 in absolute faults per system and in percentage of faults encountered. The data show a clear improvement in the reliability of hardware and maintenance. Disks in 1985 needed yearly service by Tandem, but they were replaced by disks that needed no scheduled maintenance. Shrinking numbers of chips and connectors per system plus software's ability to tolerate hardware faults reduced hardware's contribution to only 7% of failures by 1989. Moreover, when hardware was at fault, software embedded in the hardware device (firmware) was often the culprit. The data indicate that software in

Component	Total in system	Total failed	Percentage failed
SCSI controller	44	1	2.3%
SCSI cable	39	1	2.6%
SCSI disk	368	7	1.9%
IDE/ATA disk	24	6	25.0%
Disk enclosure—backplane	46	13	28.3%
Disk enclosure—power supply	92	3	3.3%
Ethernet controller	20	1	5.0%
Ethernet switch	2	1	50.0%
Ethernet cable	42	1	2.3%
CPU/motherboard	20	0	0%

Figure 6.6 Failures of components in Tertiary Disk over 18 months of operation. For each type of component, the table shows the total number in the system, the number that failed, and the percentage failure rate. Disk enclosures have two entries in the table because they had two types of problems: backplane integrity failures and power supply failures. Since each enclosure had two power supplies, a power supply failure did not affect availability. This cluster of 20 PCs, contained in seven 7-foot-high, 19-inch-wide racks, hosts 368 8.4 GB, 7200 RPM, 3.5-inch IBM disks. The PCs are P6-200 MHz with 96 MB of DRAM each. They ran FreeBSD 3.0, and the hosts are connected via switched 100 Mbit/sec Ethernet. All SCSI disks are connected to two PCs via double-ended SCSI chains to support RAID 1. The primary application is called the Zoom Project, which in 1998 was the world's largest art image database, with 72,000 images. See Talagala et al. [2000].

1989 was the major source of reported outages (62%), followed by system operations (15%).

The problem with any such statistics is that these data only refer to what is reported; for example, environmental failures due to power outages were not reported to Tandem because they were seen as a local problem. Data on operation faults is very difficult to collect because it relies on the operators to report personal mistakes, which may affect the opinion of their managers, which in turn can affect job security and pay raises. Gray believes both environmental faults and operator faults are underreported. His study concluded that achieving higher availability requires improvement in software quality and software fault tolerance, simpler operations, and tolerance of operational faults.

Other Studies of the Role of Operators in Dependability

While Tertiary Disk and Tandem are storage-oriented dependability studies, we need to look outside storage to find better measurements on the role of humans in failures. Murphy and Gent [1995] tried to improve the accuracy of data on operator faults by having the system automatically prompt the operator on each

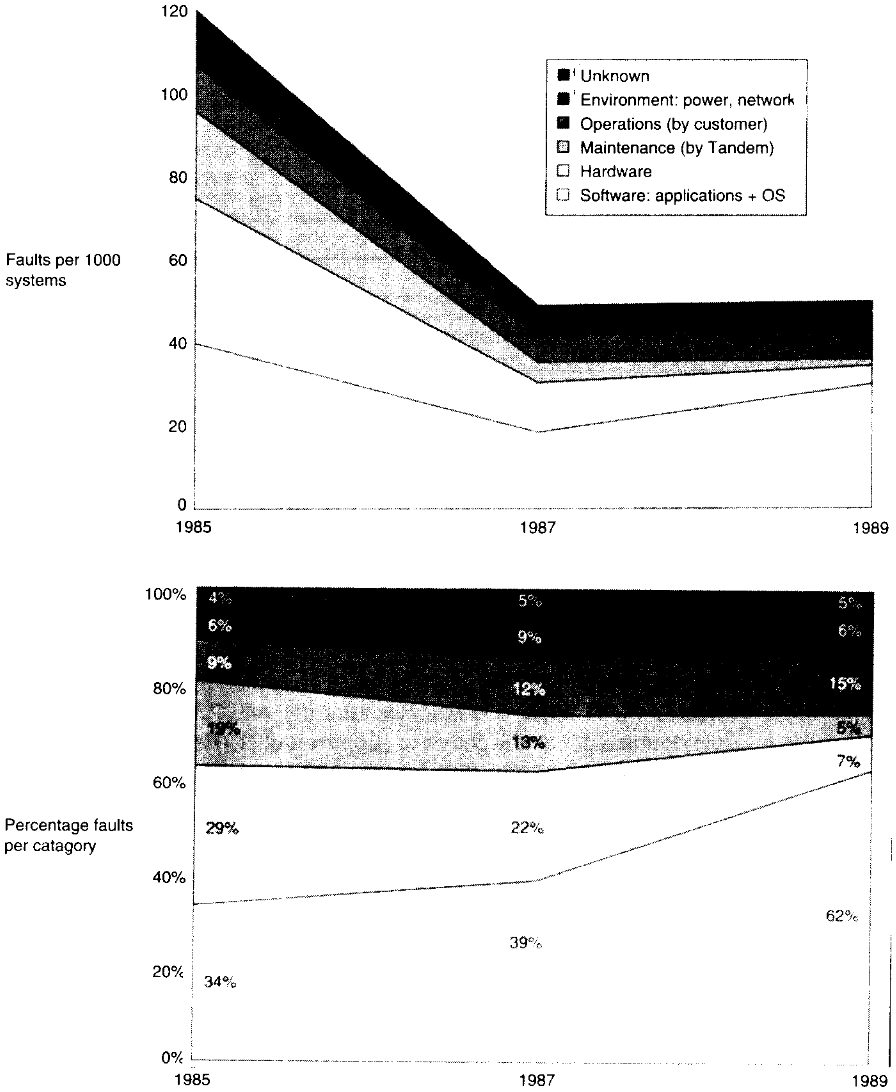


Figure 6.7 Faults in Tandem between 1985 and 1989. Gray [1990] collected these data for the fault-tolerant Tandem computers based on reports of component failures by customers.

boot for the reason for that reboot. They classified consecutive crashes to the same fault as operator fault and included operator actions that directly resulted in crashes, such as giving parameters bad values, bad configurations, and bad application installation. Although they believe operator error is still underreported, they did get more accurate information than did Gray, who relied on a form that the operator filled out and then sent up the management chain. The hardware/operating system went from causing 70% of the failures in VAX systems in 1985 to 28% in 1993, and failures due to operators rose from 15% to 52% in that same period. Murphy and Gent expected managing systems to be the primary dependability challenge in the future.

The final set of data comes from the government. The Federal Communications Commission (FCC) requires that all telephone companies submit explanations when they experience an outage that affects at least 30,000 people or lasts 30 minutes. These detailed disruption reports do not suffer from the self-reporting problem of earlier figures, as investigators determine the cause of the outage rather than operators of the equipment. Kuhn [1997] studied the causes of outages between 1992 and 1994, and Enriquez [2001] did a follow-up study for the first half of 2001. Although there was a significant improvement in failures due to overloading of the network over the years, failures due to humans increased, from about one-third to two-thirds of the customer-outage minutes.

These four examples and others suggest that the primary cause of failures in large systems today is faults by human operators. Hardware faults have declined due to a decreasing number of chips in systems and fewer connectors. Hardware dependability has improved through fault tolerance techniques such as memory ECC and RAID. At least some operating systems are considering reliability implications before adding new features, so in 2006 the failures largely occurred elsewhere.

Although failures may be initiated due to faults by operators, it is a poor reflection on the state of the art of systems that the process of maintenance and upgrading are so error prone. Most storage vendors claim today that customers spend much more on managing storage over its lifetime than they do on purchasing the storage. Thus, the challenge for dependable storage systems of the future is either to tolerate faults by operators or to avoid faults by simplifying the tasks of system administration. Note that RAID 6 allows the storage system to survive even if the operator mistakenly replaces a good disk.

We have now covered the bedrock issue of dependability, giving definitions, case studies, and techniques to improve it. The next step in the storage tour is performance.

6.4 I/O Performance, Reliability Measures, and Benchmarks

I/O performance has measures that have no counterparts in 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, namely, 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 6.8 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.

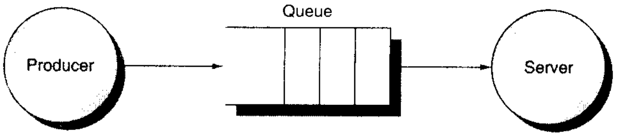


Figure 6.8 The traditional producer-server model of response time and throughput. Response time begins when a task is placed in the buffer and ends when it is completed by the server. Throughput is the number of tasks completed by the server in unit time.

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, so an empty buffer shrinks it.

Another measure of I/O performance is the interference of I/O with processor 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 much longer a process will take because of I/O for another process.

Throughput versus Response Time

Figure 6.9 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.

How does the architect balance these conflicting demands? If the computer is interacting with human beings, Figure 6.10 suggests an answer. An interaction, or *transaction*, with a computer is divided into three parts:

1. *Entry time*—The time for the user to enter the command.
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. The results in Figure 6.10 show that 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. Although this study is 20 years old, response times are often still much slower than 1 second, even if

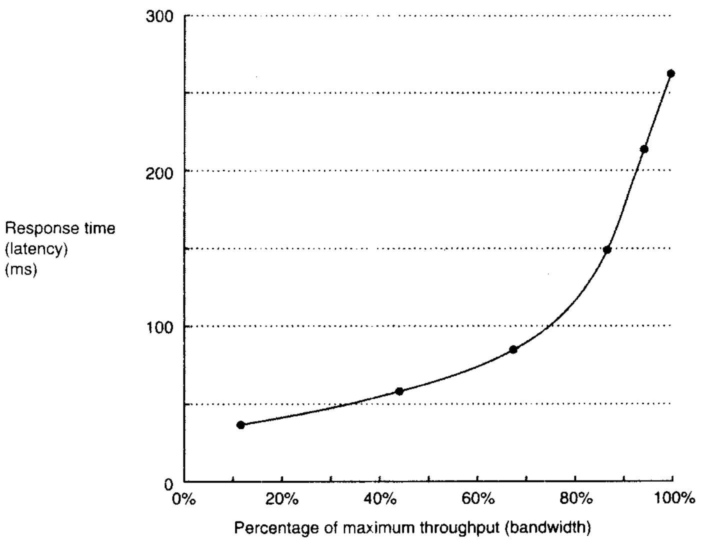


Figure 6.9 Throughput versus response time. Latency is normally reported as response time. Note that the minimum response time achieves only 11% of the throughput, while the response time for 100% throughput takes seven times the minimum response time. Note that the independent variable in this curve is implicit: to trace the curve, you typically vary load (concurrency). Chen et al. [1990] collected these data for an array of magnetic disks.

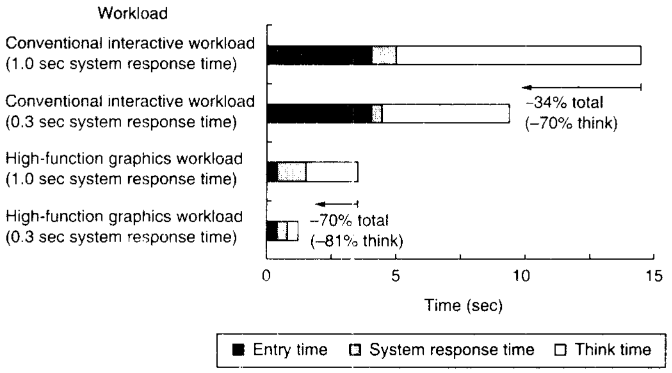


Figure 6.10 A user transaction with an interactive computer divided into entry time, system response time, and user think time for a conventional system and graphics system. The entry times are the same, independent of system response time. The entry time was 4 seconds for the conventional system and 0.25 seconds for the graphics system. Reduction in response time actually decreases transaction time by more than just the response time reduction. (From Brady [1986].)

I/O benchmark	Response time restriction	Throughput metric
TPC-C: Complex Query OLTP	≥ 90% of transaction must meet response time limit; 5 seconds for most types of transactions	new order transactions per minute
TPC-W: Transactional Web benchmark	≥ 90% of Web interactions must meet response time limit; 3 seconds for most types of Web interactions	Web interactions per second
SPECsfs97	average response time ≤ 40 ms	NFS operations per second

Figure 6.11 Response time restrictions for three I/O benchmarks.

processors are 1000 times faster. Examples of long delays include starting an application on a desktop PC due to many disk I/Os, or network delays when clicking on Web links.

To reflect the importance of response time to user productivity, I/O benchmarks also address the response time versus throughput trade-off. Figure 6.11 shows the response time bounds for three I/O benchmarks. They report maximum throughput given either that 90% of response times must be less than a limit or that the average response time must be less than a limit.

Let's next look at these benchmarks in more detail.

Transaction-Processing Benchmarks

Transaction processing (TP, or OLTP for online 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. Suppose, for example, a bank's computer fails when a customer tries to withdraw money from an ATM. 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.

As mentioned in Chapter 1, 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 [Anon. et al. 1985]. This report led to the *Transaction Processing Council*, which in turn has led to eight benchmarks since its founding. Figure 6.12 summarizes these benchmarks.

Let's describe TPC-C to give a flavor of these benchmarks. TPC-C uses a database to simulate an order-entry environment of a wholesale supplier, including entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. It runs five concurrent transactions of varying complexity, and the database includes nine tables with a scalable range of records and customers. TPC-C is measured in transac-

Benchmark	**Data size (GB)**	**Performance metric**	**Date of first results**
A: debit credit (retired)	0.1–10	transactions per second	July 1990
B: batch debit credit (retired)	0.1–10	transactions per second	July 1991
C: complex query OLTP	100–3000 (minimum 0.07 * TPM)	new order transactions per minute (TPM)	September 1992
D: decision support (retired)	100, 300, 1000	queries per hour	December 1995
H: ad hoc decision support	100, 300, 1000	queries per hour	October 1999
R: business reporting decision support (retired)	1000	queries per hour	August 1999
W: transactional Web benchmark	≈ 50, 500	Web interactions per second	July 2000
App: application server and Web services benchmark	≈ 2500	Web service interactions per second (SIPS)	June 2005

Figure 6.12 Transaction Processing Council benchmarks. The summary results include both the performance metric and the price-performance of that metric. TPC-A, TPC-B, TPC-D, and TPC-R were retired.

tions per minute (tpmC) and in price of system, including hardware, software, and three years of maintenance support. Figure 1.16 on page 46 in Chapter 1 describes the top systems in performance and cost-performance for TPC-C.

These TPC benchmarks were the first—and in some cases still the only ones—that have these unusual characteristics:

- *Price is included with the benchmark results.* The cost of hardware, software, and maintenance agreements is included in a submission, which enables evaluations based on price-performance as well as high performance.
- *The data set generally must scale in size as the throughput increases.* The benchmarks are trying to model real systems, in which the demand on the system and the size of the data stored in it increase together. It makes no sense, for example, to have thousands of people per minute access hundreds of bank accounts.
- *The benchmark results are audited.* Before results can be submitted, they must be approved by a certified TPC auditor, who enforces the TPC rules that try to make sure that only fair results are submitted. Results can be challenged and disputes resolved by going before the TPC.
- *Throughput is the performance metric, but response times are limited.* For example, with TPC-C, 90% of the New-Order transaction response times must be less than 5 seconds.
- *An independent organization maintains the benchmarks.* Dues collected by TPC pay for an administrative structure including a Chief Operating Office. This organization settles disputes, conducts mail ballots on approval of changes to benchmarks, holds board meetings, and so on.

SPEC System-Level File Server, Mail, and Web Benchmarks

The SPEC benchmarking effort is best known for its characterization of processor performance, but it has created benchmarks for file servers, mail servers, and Web servers.

Seven companies agreed on a synthetic benchmark, called SFS, to evaluate systems running the Sun Microsystems network file service (NFS). This benchmark was upgraded to SFS 3.0 (also called SPEC SFS97_R1) to include support for NFS version 3, using TCP in addition to UDP as the transport protocol, and making the mix of operations more realistic. Measurements on NFS systems led to a synthetic mix of reads, writes, and file operations. 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-C, SFS scales the amount of data stored 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 40 ms. Figure 6.13 shows average response time versus throughput for two NetApp systems. Unfortunately, unlike the TPC benchmarks, SFS does not normalize for different price configurations.

SPECMail is a benchmark to help evaluate performance of mail servers at an Internet service provider. SPECMail2001 is based on the standard Internet protocols SMTP and POP3, and it measures throughput and user response time while scaling the number of users from 10,000 to 1,000,000.

SPECWeb is a benchmark for evaluating the performance of World Wide Web servers, measuring number of simultaneous user sessions. The SPECWeb2005

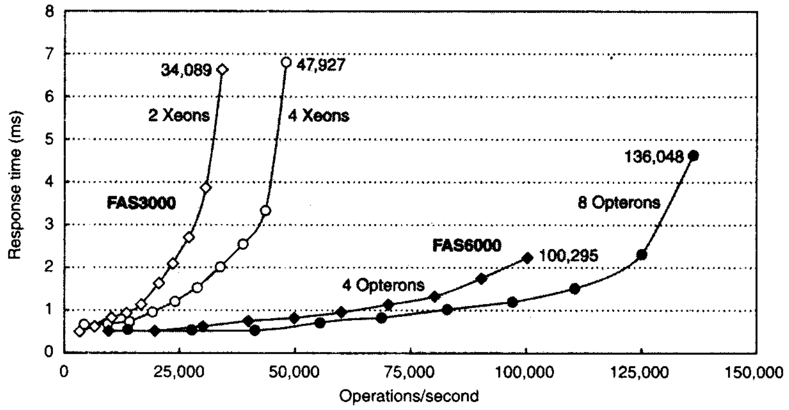


Figure 6.13 SPEC SFS97_R1 performance for the NetApp FAS3050c NFS servers in two configurations. Two processors reached 34,089 operations per second and four processors did 47,927. Reported in May 2005, these systems used the Data ONTAP 7.0.1R1 operating system, 2.8 GHz Pentium Xeon microprocessors, 2 GB of DRAM per processor, 1 GB of nonvolatile memory per system, and 168 15K RPM, 72 GB, fibre channel disks. These disks were connected using two or four QLogic ISP-2322 FC disk controllers.

workload simulates accesses to a Web service provider, where the server supports home pages for several organizations. It has three workloads: Banking (HTTPS), E-commerce (HTTP and HTTPS), and Support (HTTP).

Examples of Benchmarks of Dependability

The TPC-C benchmark does in fact have a dependability requirement. The benchmarked system must be able to handle a single disk failure, which means in practice that all submitters are running some RAID organization in their storage system.

Efforts that are more recent have focused on the effectiveness of fault tolerance in systems. Brown and Patterson [2000] propose that availability be measured by examining the variations in system quality-of-service metrics over time as faults are injected into the system. For a Web server the obvious metrics are performance (measured as requests satisfied per second) and degree of fault tolerance (measured as the number of faults that can be tolerated by the storage subsystem, network connection topology, and so forth).

The initial experiment injected a single fault—such as a write error in disk sector—and recorded the system's behavior as reflected in the quality-of-service metrics. The example compared software RAID implementations provided by Linux, Solaris, and Windows 2000 Server. SPECWeb99 was used to provide a workload and to measure performance. To inject faults, one of the SCSI disks in the software RAID volume was replaced with an emulated disk. It was a PC running software using a SCSI controller that appears to other devices on the SCSI bus as a disk. The disk emulator allowed the injection of faults. The faults injected included a variety of transient disk faults, such as correctable read errors, and permanent faults, such as disk media failures on writes.

Figure 6.14 shows the behavior of each system under different faults. The two top graphs show Linux (on the left) and Solaris (on the right). As RAID systems can lose data if a second disk fails before reconstruction completes, the longer the reconstruction (MTTR), the lower the availability. Faster reconstruction implies decreased application performance, however, as reconstruction steals I/O resources from running applications. Thus, there is a policy choice between taking a performance hit during reconstruction, or lengthening the window of vulnerability and thus lowering the predicted MTTF.

Although none of the tested systems documented their reconstruction policies outside of the source code, even a single fault injection was able to give insight into those policies. The experiments revealed that both Linux and Solaris initiate automatic reconstruction of the RAID volume onto a hot spare when an active disk is taken out of service due to a failure. Although Windows supports RAID reconstruction, the reconstruction must be initiated manually. Thus, without human intervention, a Windows system that did not rebuild after a first failure remains susceptible to a second failure, which increases the window of vulnerability. It does repair quickly once told to do so.

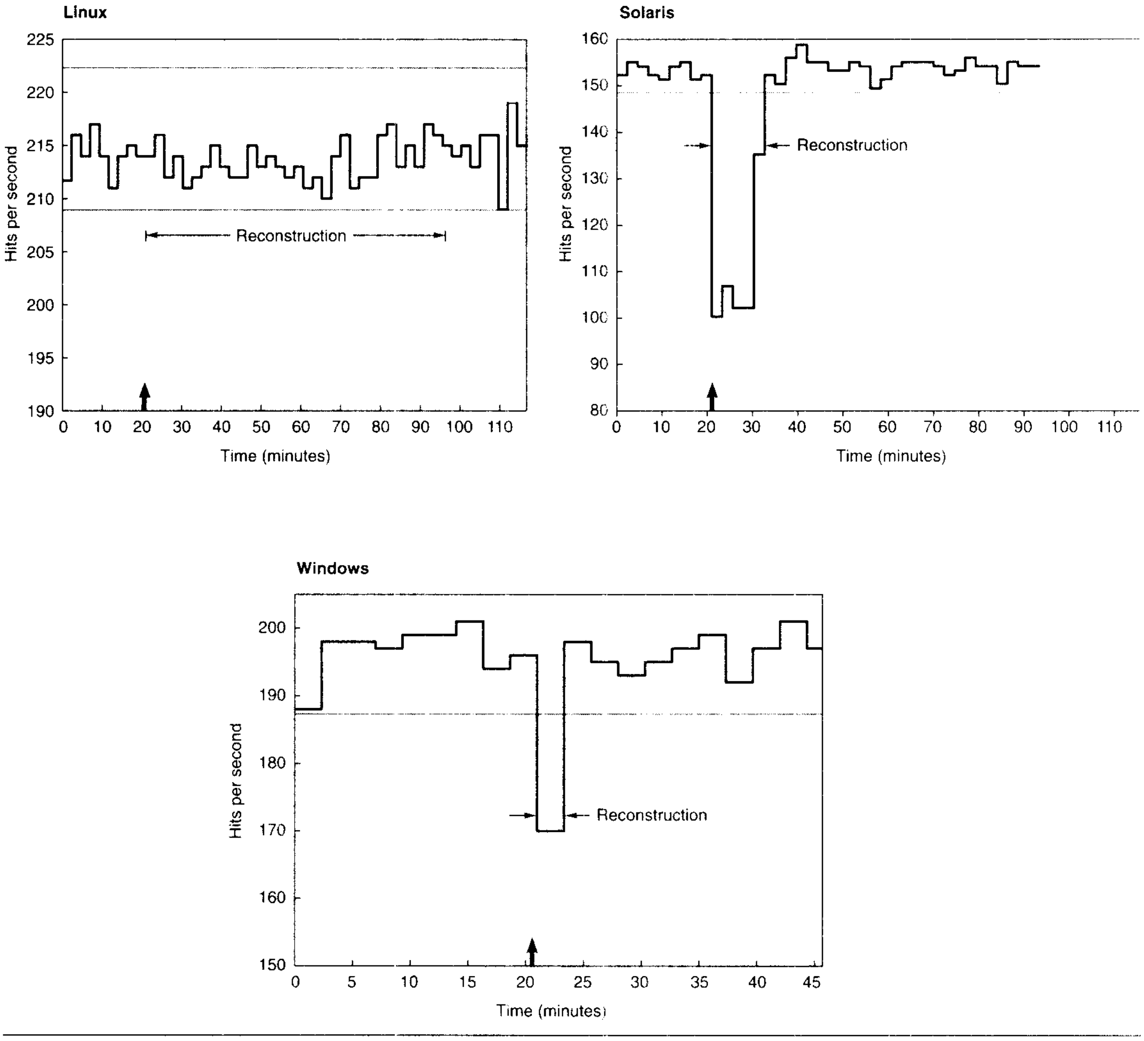


Figure 6.14 Availability benchmark for software RAID systems on the same computer running Red Hat 6.0 Linux, Solaris 7, and Windows 2000 operating systems. Note the difference in philosophy on speed of reconstruction of Linux versus Windows and Solaris. The *y*-axis is behavior in hits per second running SPECWeb99. The arrow indicates time of fault insertion. The lines at the top give the 99% confidence interval of performance before the fault is inserted. A 99% confidence interval means that if the variable is outside of this range, the probability is only 1% that this value would appear.

The fault injection experiments also provided insight into other availability policies of Linux, Solaris, and Windows 2000 concerning automatic spare utilization, reconstruction rates, transient errors, and so on. Again, no system documented their policies.

In terms of managing transient faults, the fault injection experiments revealed that Linux's software RAID implementation takes an opposite approach than do the RAID implementations in Solaris and Windows. The Linux implementation is paranoid—it would rather shut down a disk in a controlled manner at the first error, rather than wait to see if the error is transient. In contrast, Solaris and Windows are more forgiving—they ignore most transient faults with the expectation that they will not recur. Thus, these systems are substantially more robust to transients than the Linux system. Note that both Windows and Solaris do log the transient faults, ensuring that the errors are reported even if not acted upon. When faults were permanent, the systems behaved similarly.

6.5 A Little Queuing Theory

In processor design, we have simple back-of-the-envelope calculations of performance associated with the CPI formula in Chapter 1, or we can use full scale simulation for greater accuracy at greater cost. In I/O systems, we also have a best-case analysis as a back-of-the-envelope calculation. Full-scale simulation is also much more accurate and much more work to calculate expected performance.

With I/O systems, however, we also have a mathematical tool to guide I/O design that is a little more work and much more accurate than best-case analysis, but much less work than full-scale simulation. Because of the probabilistic nature of I/O events and because of sharing of I/O resources, we can give a set of simple theorems that will help calculate response time and throughput of an entire I/O system. This helpful field is called *queuing theory*. Since there are many books and courses on the subject, this section serves only as a first introduction to the topic. However, even this small amount can lead to better design of I/O systems.

Let's start with a black-box approach to I/O systems, as in Figure 6.15. In our example, the processor is making I/O requests that arrive at the I/O device, and the requests "depart" when the I/O device fulfills them.

We are usually interested in the long term, or steady state, of a system rather than in the initial start-up conditions. Suppose we weren't. Although there is a mathematics that helps (Markov chains), except for a few cases, the only way to solve the resulting equations is simulation. Since the purpose of this section is to show something a little harder than back-of-the-envelope calculations but less

Figure 6.15 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 system must equal the number of tasks leaving the system. This *flow-balanced* state is necessary but not sufficient for steady state. If the system has been observed or measured for a sufficiently long time and mean waiting times stabilize, then we say that the system has reached steady state.

than simulation, we won't cover such analyses here. (See the references in Appendix K for more details.)

Hence, in this section we make the simplifying assumption that we are evaluating systems with multiple independent requests for I/O service that are in equilibrium: the input rate must be equal to the output rate. We also assume there is a steady supply of tasks independent for how long they wait for service. In many real systems, such as TPC-C, the task consumption rate is determined by other system characteristics, such as memory capacity.

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. Note that the arrival rate and the response time must use the same time unit; inconsistency in time units is a common cause of errors.

Let's try to derive Little's Law. Assume we observe a system for $\text{Time}_{\text{observe}}$ minutes. During that observation, we record how long it took each task to be serviced, and then sum those times. The number of tasks completed during $\text{Time}_{\text{observe}}$ is $\text{Number}_{\text{task}}$, and the sum of the times each task spends in the system is $\text{Time}_{\text{accumulated}}$. Note that the tasks can overlap in time, so $\text{Time}_{\text{accumulated}} \geq \text{Time}_{\text{observed}}$. Then

$$\text{Mean number of tasks in system} = \frac{\text{Time}_{\text{accumulated}}}{\text{Time}_{\text{observe}}}$$

$$\text{Mean response time} = \frac{\text{Time}_{\text{accumulated}}}{\text{Number}_{\text{tasks}}}$$

$$\text{Arrival rate} = \frac{\text{Number}_{\text{tasks}}}{\text{Time}_{\text{observe}}}$$

Algebra lets us split the first formula:

$$\frac{\text{Time}_{\text{accumulated}}}{\text{Time}_{\text{observe}}} = \frac{\text{Time}_{\text{accumulated}}}{\text{Number}_{\text{tasks}}} \times \frac{\text{Number}_{\text{tasks}}}{\text{Time}_{\text{observe}}}$$

If we substitute the three definitions above into this formula, and swap the resulting two terms on the right-hand side, we get Little's Law:

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

This simple equation is surprisingly powerful, as we shall see.

If we open the black box, we see Figure 6.16. The area where the tasks accumulate, waiting to be serviced, is called the *queue,* or *waiting line.* The device performing the requested service is called the *server.* Until we get to the last two pages of this section, we assume a single server.

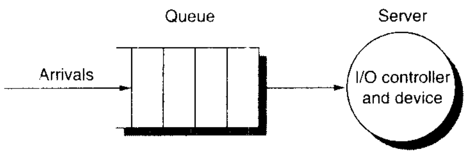


Figure 6.16 The single-server model for this section. In this situation, an I/O request "departs" by being completed by the server.

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

- $\text{Time}_{\text{server}}$—Average time to service a task; average service rate is $1/\text{Time}_{\text{server}}$, traditionally represented by the symbol μ in many queuing 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, which is 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 queuing 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, which is 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 $\text{Time}_{\text{system}} = \text{Time}_{\text{queue}} + \text{Time}_{\text{server}}$.

The mean number of tasks in service ($\text{Length}_{\text{server}}$) is simply Arrival rate × $\text{Time}_{\text{server}}$, which is Little's Law. Server utilization is simply the mean number of tasks being serviced divided by the service rate. For a single server, the service rate is $1/\text{Time}_{\text{server}}$. Hence, server utilization (and, in this case, the mean number of tasks per server) is simply

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

Service utilization must be between 0 and 1; otherwise, there would be more tasks arriving than could be serviced, violating our assumption that the system is in equilibrium. Note that this formula is just a restatement of Little's Law. Utilization is also called *traffic intensity* and is represented by the symbol ρ in many queuing theory texts.

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

Answer Using the equation above, with 10 ms represented as 0.01 seconds; we get:

$$\text{Server utilization} = \text{Arrival rate} \times \text{Time}_{\text{server}} = \frac{50}{\text{sec}} \times 0.01\,\text{sec} = 0.50$$

Therefore, the I/O system utilization is 0.5.

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:

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

That is, the time in the queue is the number of tasks in the queue times the mean service time plus the time it takes the server to complete whatever task is being serviced when a new task arrives. (There is one more restriction about the arrival of tasks, which we reveal on page 384.)

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.

Poisson Distribution of Random Variables

To estimate the last component of the formula 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 may know the probability of all possible values.

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.

One way to characterize the distribution of values of a random variable with discrete values 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 have two choices. Either we need a curve to plot the values over the full range, so that we can estimate accurately the value, or we need a very fine time unit so that we get a very large number of buckets to estimate time accurately. For example, a histogram can be built of disk service times measured in intervals of 10 µs although disk service times are truly continuous.

Hence, to be able to solve the last part of the previous equation we need to characterize the distribution of this random variable. The mean time and some measure of the variance are sufficient for that characterization.

For the first term, we use the *weighted arithmetic mean time.* Let's first assume that after measuring the number of occurrences, say, n_i, of tasks, you could compute frequency of occurrence of task i:

$$f_i = \frac{n_i}{\left(\sum_{i=1}^{n} n_i\right)}$$

Then weighted arithmetic mean is

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

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, as it will help us with characterizing the probability distribution. Given the weighted arithmetic mean, the variance can be calculated as

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

It is important to remember the units when computing variance. Let's assume the distribution is of time. If time is about 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^2:

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

We can solve for C, the coefficient of variance, as

$$C = \frac{\sqrt{\text{Variance}}}{\text{Weighted arithmetic mean time}} = \frac{\text{Standard deviation}}{\text{Weighted arithmetic mean time}}$$

We are trying to characterize random events, but to be able to predict performance we need a distribution of random events where the mathematics is tractable. The most popular such distribution is the *exponential distribution,* which has a C value of 1.

Note that we are using a constant to characterize variability about the mean. The invariance of C over time reflects the property that the history of events has no impact on the probability of an event occurring now. This forgetful property is called *memoryless,* and this property is an important assumption used to predict behavior using these models. (Suppose this memoryless property did not exist; then we would have to worry about the exact arrival times of requests relative to each other, which would make the mathematics considerably less tractable!)

One of the most widely used exponential distributions is called a *Poisson distribution,* named after the mathematician Simeon Poisson. It is used to characterize random events in a given time interval and has several desirable mathematical properties. The Poisson distribution is described by the following equation (called the probability mass function):

$$\text{Probability}(k) = \frac{e^{-a} \times a^{k}}{k!}$$

where a = Rate of events × Elapsed time. If interarrival times are exponentially distributed and we use arrival rate from above for rate of events, the number of arrivals in a time interval t is a *Poisson process,* which has the Poisson distribution with a = Arrival rate × t. As mentioned on page 382, the equation for $\text{Time}_{\text{server}}$ has another restriction on task arrival: It holds only for Poisson processes.

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,* which again assumes Poisson arrivals:

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

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 standard deviation is 0 and so C is 0. The average residual service time is then just half the average service time, as we would expect. If the distribution is random and it is Poisson, then C is 1 and the average residual service time equals the weighted arithmetic mean time.

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 ($\text{Time}_{\text{server}}$) and server utilization.

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 × 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 × $\text{Time}_{\text{queue}}$ yields

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

Since this section is concerned with exponential distributions, C^2 is 1. Thus

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

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

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

Rearranging terms and simplifying gives us the desired equation:

$$\begin{aligned}\text{Time}_{\text{queue}} &= \text{Server utilization} \times \text{Time}_{\text{server}} + \text{Server utilization} \times \text{Time}_{\text{queue}} \\ \text{Time}_{\text{queue}} - \text{Server utilization} \times \text{Time}_{\text{queue}} &= \text{Server utilization} \times \text{Time}_{\text{server}} \\ \text{Time}_{\text{queue}} \times (1 - \text{Server utilization}) &= \text{Server utilization} \times \text{Time}_{\text{server}} \\ \text{Time}_{\text{queue}} &= \text{Time}_{\text{server}} \times \frac{\text{Server utilization}}{(1 - \text{Server utilization})}\end{aligned}$$

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

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

If we substitute for $\text{Time}_{\text{queue}}$ from above, we get

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

Since Arrival rate × $\text{Time}_{\text{server}}$ = Server utilization, we can simplify further:

$$\text{Length}_{\text{queue}} = \text{Server utilization} \times \frac{\text{Server utilization}}{(1 - \text{Server utilization})} = \frac{\text{Server utilization}^2}{(1 - \text{Server utilization})}$$

This relates number of items in queue to service utilization.

Example For the system in the example on page 382, which has a server utilization of 0.5, what is the mean number of I/O requests in the queue?

Answer Using the equation above,

$$\text{Length}_{\text{queue}} = \frac{\text{Server utilization}^2}{(1 - \text{Server utilization})} = \frac{0.5^2}{(1 - 0.5)} = \frac{0.25}{0.50} = 0.5$$

Therefore, there are 0.5 requests on average in the queue.

As mentioned earlier, these equations and this section 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.

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 more sophisticated mathematics. In computer systems, we commonly predict the future from the past; one example is least-recently used block replacement (see Chapter 5). Hence, the distinction between measurements and predicted distributions is often blurred; 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, which characterizes the arrival rate mentioned earlier.
- The number of sources of requests is unlimited. (This is called an *infinite population model* in queuing theory; finite population models are used when arrival rates vary with the number of jobs already in the system.)
- The server can start on the next job immediately after finishing the prior one.
- There is no limit to the length of the queue, and it follows the first in, first out order discipline, so all tasks in line must be completed.
- There is one server.

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

M = exponentially random request arrival ($C^2 = 1$), with M standing for A. A. Markov, the mathematician who defined and analyzed the memoryless processes mentioned earlier

M = exponentially random service time ($C^2 = 1$), with M again for Markov

1 = single server

The M/M/1 model is a simple and widely used model.

The assumption of exponential distribution is commonly used in queuing examples for three reasons—one good, one fair, and one bad. The good reason is that a superposition 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 fair reason is that when variability is unclear, an exponential distribution with intermediate variability ($C = 1$) is a safer guess than low variability ($C \approx 0$) or high variability (large C). 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 40 disk I/Os per second, these requests are exponentially distributed, and the average service time of an older disk 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 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 40.

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

The server utilization is then

$$\text{Server utilization} = \text{Arrival rate} \times \text{Time}_{\text{server}} = 40 \times 0.02 = 0.8$$

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

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

$$= 20 \text{ ms} \times \frac{0.8}{1 - 0.8} = 20 \times \frac{0.8}{0.2} = 20 \times 4 = 80 \text{ ms}$$

The average response time is

$$\text{Time system} = \text{Time}_{\text{queue}} + \text{Time}_{\text{server}} = 80 + 20 \text{ ms} = 100 \text{ ms}$$

Thus, on average we spend 80% of our time waiting in the queue!

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

$$\text{Server utilization} = \text{Arrival rate} \times \text{Time}_{\text{server}} = 40 \times 0.01 = 0.4$$

The formula for the average time spent waiting in line:

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

$$= 10 \text{ ms} \times \frac{0.4}{1 - 0.4} = 10 \times \frac{0.4}{0.6} = 10 \times \frac{2}{3} = 6.7 \text{ ms}$$

The average response time is 10 + 6.7 ms or 16.7 ms, 6.0 times faster than the old response time even though the new service time is only 2.0 times faster.

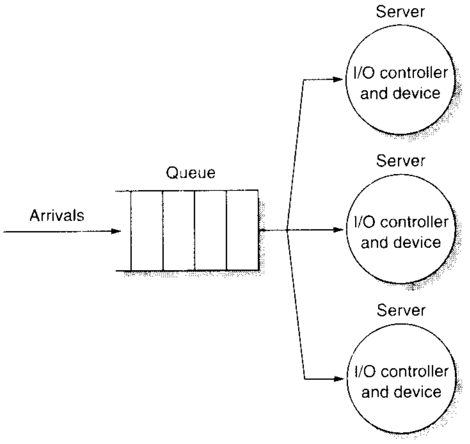


Figure 6.17 The M/M/m multiple-server model.

Thus far, we have been assuming a single server, such as a single disk. Many real systems have multiple disks and hence could use multiple servers, as in Figure 6.17. Such a system is called an *M/M/m* model in queuing theory.

Let's give the same formulas for the M/M/m queue, using $N_{servers}$ to represent the number of servers. The first two formulas are easy:

$$\text{Utilization} = \frac{\text{Arrival rate} \times \text{Time}_{server}}{N_{servers}}$$

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

The time waiting in the queue is

$$\text{Time}_{queue} = \text{Time}_{server} \times \frac{P_{tasks \geq N_{servers}}}{N_{servers} \times (1 - \text{Utilization})}$$

This formula is related to the one for M/M/1, except we replace utilization of a single server with the probability that a task will be queued as opposed to being immediately serviced, and divide the time in queue by the number of servers. Alas, calculating the probability of jobs being in the queue is much more complicated when there are $N_{servers}$. First, the probability that there are no tasks in the system is

$$\text{Prob}_{0\ tasks} = \left[1 + \frac{(N_{servers} \times \text{Utilization})^{N_{servers}}}{N_{servers}! \times (1 - \text{Utilization})} + \sum_{n=1}^{N_{servers}-1} \frac{(N_{servers} \times \text{Utilization})^{n}}{n!} \right]^{-1}$$

Then the probability there are as many or more tasks than we have servers is

$$\text{Prob}_{tasks \geq N_{servers}} = \frac{N_{servers} \times \text{Utilization}^{N_{servers}}}{N_{servers}! \times (1 - \text{Utilization})} \times \text{Prob}_{0\ tasks}$$

Note that if N_{servers} is 1, $\text{Prob}_{\text{task} \geq N_{\text{servers}}}$ simplifies back to Utilization, and we get the same formula as for M/M/1. Let's try an example.

Example Suppose instead of a new, faster disk, we add a second slow disk and duplicate the data so that reads can be serviced by either disk. Let's assume that the requests are all reads. Recalculate the answers to the earlier questions, this time using an M/M/m queue.

Answer The average utilization of the two disks is then

$$\text{Server utilization} = \frac{\text{Arrival rate} \times \text{Time}_{\text{server}}}{N_{\text{servers}}} = \frac{40 \times 0.02}{2} = 0.4$$

We first calculate the probability of no tasks in the queue:

$$\begin{aligned}
\text{Prob}_{0\ \text{tasks}} &= \left[1 + \frac{(2 \times \text{Utilization})^2}{2! \times (1 - \text{Utilization})} + \sum_{n=1}^{1} \frac{(2 \times \text{Utilization})^n}{n!}\right]^{-1} \\
&= \left[1 + \frac{(2 \times 0.4)^2}{2 \times (1 - 0.4)} + (2 \times 0.4)\right]^{-1} = \left[1 + \frac{0.640}{1.2} + 0.800\right]^{-1} \\
&= [1 + 0.533 + 0.800]^{-1} = 2.333^{-1}
\end{aligned}$$

We use this result to calculate the probability of tasks in the queue:

$$\begin{aligned}
\text{Prob}_{\text{tasks} \geq N_{\text{servers}}} &= \frac{2 \times \text{Utilization}^2}{2! \times (1 - \text{Utilization})} \times \text{Prob}_{0\ \text{tasks}} \\
&= \frac{(2 \times 0.4)^2}{2 \times (1 - 0.4)} \times 2.333^{-1} = \frac{0.640}{1.2} \times 2.333^{-1} \\
&= 0.533/2.333 = 0.229
\end{aligned}$$

Finally, the time waiting in the queue:

$$\begin{aligned}
\text{Time}_{\text{queue}} &= \text{Time}_{\text{server}} \times \frac{\text{Prob}_{\text{tasks} \geq N_{\text{servers}}}}{N_{\text{servers}} \times (1 - \text{Utilization})} \\
&= 0.020 \times \frac{0.229}{2 \times (1 - 0.4)} = 0.020 \times \frac{0.229}{1.2} \\
&= 0.020 \times 0.190 = 0.0038
\end{aligned}$$

The average response time is 20 + 3.8 ms or 23.8 ms. For this workload, two disks cut the queue waiting time by a factor of 21 over a single slow disk and a factor of 1.75 versus a single fast disk. The mean service time of a system with a single fast disk, however, is still 1.4 times faster than one with two disks since the disk service time is 2.0 times faster.

It would be wonderful if we could generalize the M/M/m model to multiple queues and multiple servers, as this step is much more realistic. Alas, these models are very hard to solve and to use, and so we won't cover them here.

6.6 Crosscutting Issues

Point-to-Point Links and Switches Replacing Buses

Point-to-point links and switches are increasing in popularity as Moore's Law continues to reduce the cost of components. Combined with the higher I/O bandwidth demands from faster processors, faster disks, and faster local area networks, the decreasing cost advantage of buses means the days of buses in desktop and server computers are numbered. This trend started in high-performance computers in the last edition of the book, and by 2006 has spread itself throughout the storage. Figure 6.18 shows the old bus-based standards and their replacements.

The number of bits and bandwidth for the new generation is per direction, so they double for both directions. Since these new designs use many fewer wires, a common way to increase bandwidth is to offer versions with several times the number of wires and bandwidth.

Block Servers versus Filers

Thus far, we have largely ignored the role of the operating system in storage. In a manner analogous to the way compilers use an instruction set, operating systems determine what I/O techniques implemented by the hardware will actually be used. The operating system typically provides the file abstraction on top of blocks stored on the disk. The terms *logical units, logical volumes,* and *physical volumes* are related terms used in Microsoft and UNIX systems to refer to subset collections of disk blocks.

A logical unit is the element of storage exported from a disk array, usually constructed from a subset of the array's disks. A logical unit appears to the server

Standard	**Width (bits)**	**Length (meters)**	**Clock rate**	**MB/sec**	**Max I/O devices**
(Parallel) ATA	8	0.5	133 MHz	133	2
Serial ATA	2	2	3 GHz	300	?
SCSI	16	12	80 MHz	320	15
Serial Attach SCSI	1	10	(DDR)	375	16,256
PCI	32/64	0.5	33/66 MHz	533	?
PCI Express	2	0.5	3 GHz	250	?

Figure 6.18 Parallel I/O buses and their point-to-point replacements. Note the bandwidth and wires are per direction, so bandwidth doubles when sending both directions.

as a single virtual "disk." In a RAID disk array, the logical unit is configured as a particular RAID layout, such as RAID 5. A physical volume is the device file used by the file system to access a logical unit. A logical volume provides a level of virtualization that enables the file system to split the physical volume across multiple pieces or to stripe data across multiple physical volumes. A logical unit is an abstraction of a disk array that presents a virtual disk to the operating system, while physical and logical volumes are abstractions used by the operating system to divide these virtual disks into smaller, independent file systems.

Having covered some of the terms for collections of blocks, the question arises, Where should the file illusion be maintained: in the server or at the other end of the storage area network?

The traditional answer is the server. It accesses storage as disk blocks and maintains the metadata. Most file systems use a file cache, so the server must maintain consistency of file accesses. The disks may be *direct attached*—found inside a server connected to an I/O bus—or attached over a storage area network, but the server transmits data blocks to the storage subsystem.

The alternative answer is that the disk subsystem itself maintains the file abstraction, and the server uses a file system protocol to communicate with storage. Example protocols are Network File System (NFS) for UNIX systems and Common Internet File System (CIFS) for Windows systems. Such devices are called *network attached storage* (NAS) devices since it makes no sense for storage to be directly attached to the server. The name is something of a misnomer because a storage area network like FC-AL can also be used to connect to block servers. The term *filer* is often used for NAS devices that only provide file service and file storage. Network Appliances was one of the first companies to make filers.

The driving force behind placing storage on the network is to make it easier for many computers to share information and for operators to maintain the shared system.

Asynchronous I/O and Operating Systems

Disks typically spend much more time in mechanical delays than in transferring data. Thus, a natural path to higher I/O performance is parallelism, trying to get many disks to simultaneously access data for a program.

The straightforward approach to I/O is to request data and then start using it. The operating system then switches to another process until the desired data arrive, and then the operating system switches back to the requesting process. Such a style is called *synchronous I/O*—the process waits until the data have been read from disk.

The alternative model is for the process to continue after making a request, and it is not blocked until it tries to read the requested data. Such *asynchronous I/O* allows the process to continue making requests so that many I/O requests can be operating simultaneously. Asynchronous I/O shares the same philosophy as caches in out-of-order CPUs, which achieve greater bandwidth by having multiple outstanding events.

6.7 Designing and Evaluating an I/O System—The Internet Archive Cluster

The art of I/O system design is to find a design that meets goals for cost, dependability, and variety of devices while avoiding bottlenecks in I/O performance and dependability. Avoiding bottlenecks means that components must be balanced between main memory and the I/O device, because performance and dependability—and hence effective cost-performance or cost-dependability—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 and networks, larger power supplies to support I/O devices, and larger cabinets.

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

1. List the different types of I/O devices to be connected to the machine, or list the standard buses and networks that the machine will support.
2. List the physical requirements for each I/O device. Requirements include size, 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. List the reliability of each I/O device.
5. Record the processor resource demands of each I/O device. This list 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 to complete I/O
    - Processor clock stalls due to waiting for I/O to finish using the memory, bus, or cache
    - Processor clock cycles to recover from an I/O activity, such as a cache flush
6. List the memory and I/O bus resource demands of each I/O device. Even when the processor is not using memory, the bandwidth of main memory and the I/O connection is limited.
7. The final step is assessing the performance and availability of the different ways to organize these I/O devices. When you can afford it, try to avoid single points of failure. Performance can only be properly evaluated with simulation, although it may be estimated using queuing theory. Reliability can be calculated assuming I/O devices fail independently and that the times to failure are

exponentially distributed. Availability can be computed from reliability by estimating MTTF for the devices, taking into account the time from failure to repair.

Given your cost, performance, and availability goals, you then select the best organization,

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 processor. For low cost, most of the cost should be the I/O devices themselves. Availability goals depend in part on the cost of unavailability to an organization.

Rather than create a paper design, let's evaluate a real system.

The Internet Archive Cluster

To make these ideas clearer, we'll estimate the cost, performance, and availability of a large storage-oriented cluster at the Internet Archive. The Internet Archive began in 1996 with the goal of making a historical record of the Internet as it changed over time. You can use the Wayback Machine interface to the Internet Archive to perform time travel to see what the Web site at a URL looked like some time in the past. In 2006 it contains over a petabyte (10^{15} bytes) and is growing by 20 terabytes (10^{12} bytes) of new data per month, so expansible storage is a requirement. In addition to storing the historical record, the same hardware is used to crawl the Web every few months to get snapshots of the Internet.

Clusters of computers connected by local area networks have become a very economical computation engine that work well for some applications. Clusters also play an important role in Internet services such the Google search engine, where the focus is more on storage than it is on computation, as is the case here.

Although it has used a variety of hardware over the years, the Internet Archive is moving to a new cluster to become more efficient in power and in floor space. The basic building block is a 1U storage node called the PetaBox GB2000 from Capricorn Technologies. In 2006 it contains four 500 GB Parallel ATA (PATA) disk drives, 512 MB of DDR266 DRAM, one 10/100/1000 Ethernet interface, and a 1 GHz C3 Processor from VIA, which executes the 80x86 instruction set. This node dissipates about 80 watts in typical configurations.

Figure 6.19 shows the cluster in a standard VME rack. Forty of the GB2000s fit in a standard VME rack, which gives the rack 80 TB of raw capacity. The 40 nodes are connected together with a 48-port 10/100 or 10/100/1000 switch, and it dissipates about 3 KW. The limit is usually 10 KW per rack in computer facilities, so it is well within the guidelines.

A petabyte needs 12 of these racks, connected by a higher-level switch that connects the Gbit links coming from the switches in each of the racks.

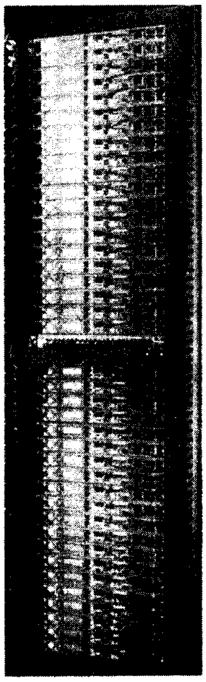


Figure 6.19 The TB-80 VME rack from Capricorn Systems used by the Internet Archive. All cables, switches, and displays are accessible from the front side, and so the back side is only used for airflow. This allows two racks to be placed back-to-back, which reduces the floor space demands in machine rooms.

Estimating Performance, Dependability, and Cost of the Internet Archive Cluster

To illustrate how to evaluate an I/O system, we'll make some guesses about the cost, performance, and reliability of the components of this cluster. We make the following assumptions about cost and performance:

- The VIA processor, 512 MB of DDR266 DRAM, ATA disk controller, power supply, fans, and enclosure costs $500.
- Each of the four 7200 RPM Parallel ATA drives holds 500 GB, has an average time seek of 8.5 ms, transfers at 50 MB/sec from the disk, and costs $375. The PATA link speed is 133 MB/sec.
- The 48-port 10/100/1000 Ethernet switch and all cables for a rack costs $3000.
- The performance of the VIA processor is 1000 MIPS.
- The ATA controller adds 0.1 ms of overhead to perform a disk I/O.
- The operating system uses 50,000 CPU instructions for a disk I/O.

- The network protocol stacks use 100,000 CPU instructions to transmit a data block between the cluster and the external world.
- The average I/O size is 16 KB for accesses to the historical record via the Wayback interface, and 50 KB when collecting a new snapshot.

Example Evaluate the cost per I/O per second (IOPS) of the 80 TB rack. Assume that every disk I/O requires an average seek and average rotational delay. Assume the workload is evenly divided among all disks and that all devices can be used at 100% of capacity; that is, the system is limited only by the weakest link, and it can operate that link at 100% utilization. Calculate for both average I/O sizes.

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 of one GB2000. The CPU I/O performance is determined by the speed of the CPU and the number of instructions to perform a disk I/O and to send it over the network:

$$\text{Maximum IOPS for CPU} = \frac{1000 \text{ MIPS}}{50{,}000 \text{ instructions per I/O} + 100{,}000 \text{ instructions per message}} = 6667 \text{ IOPS}$$

The maximum performance of the memory system is determined by the memory bandwidth and the size of the I/O transfers:

$$\text{Maximum IOPS for main memory} = \frac{266 \times 8}{16 \text{ KB per I/O}} \approx 133{,}000 \text{ IOPS}$$

$$\text{Maximum IOPS for main memory} = \frac{266 \times 8}{50 \text{ KB per I/O}} \approx 42{,}500 \text{ IOPS}$$

The Parallel ATA link performance is limited by the bandwidth and the size of the I/O:

$$\text{Maximum IOPS for the I/O bus} = \frac{133 \text{ MB/sec}}{16 \text{ KB per I/O}} \approx 8300 \text{ IOPS}$$

$$\text{Maximum IOPS for the I/O bus} = \frac{133 \text{ MB/sec}}{50 \text{ KB per I/O}} \approx 2700 \text{ IOPS}$$

Since the box has two buses, the I/O bus limits the maximum performance to no more than 18,600 IOPS for 16 KB blocks and 5400 IOPS for 50 KB blocks.

Now it's time to look at the performance of the next link in the I/O chain, the ATA controllers. The time to transfer a block over the PATA channel is

$$\text{Parallel ATA transfer time} = \frac{16 \text{ KB}}{133 \text{ MB/sec}} = 0.1 \text{ ms}$$

$$\text{Parallel ATA transfer time} = \frac{50 \text{ KB}}{133 \text{ MB/sec}} = 0.4 \text{ ms}$$

Adding the 0.1 ms ATA controller overhead means 0.2 ms to 0.5 ms per I/O, making the maximum rate per controller

$$\text{Maximum IOPS per ATA controller} = \frac{1}{0.2\text{ ms}} = 5000\text{ IOPS}$$

$$\text{Maximum IOPS per ATA controller} = \frac{1}{0.5\text{ ms}} = 2000\text{ IOPS}$$

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

$$\text{I/O time} = 8.5\text{ ms} + \frac{0.5}{7200\text{ RPM}} + \frac{16\text{ KB}}{50\text{ MB/sec}} = 8.5 + 4.2 + 0.3 = 13.0\text{ ms}$$

$$\text{I/O time} = 8.5\text{ ms} + \frac{0.5}{7200\text{ RPM}} + \frac{50\text{ KB}}{50\text{ MB/sec}} = 8.5 + 4.2 + 1.0 = 13.7\text{ ms}$$

Therefore, disk performance is

$$\text{Maximum IOPS (using average seeks) per disk} = \frac{1}{13.0\text{ ms}} \approx 77\text{ IOPS}$$

$$\text{Maximum IOPS (using average seeks) per disk} = \frac{1}{13.7\text{ ms}} \approx 73\text{ IOPS}$$

or 292–308 IOPS for the four disks.

The final link in the chain is the network that connects the computers to the outside world. The link speed determines the limit

$$\text{Maximum IOPS per 1000 Mbit Ethernet link} = \frac{1000\text{ Mbit}}{16\text{K} \times 8} = 7812\text{ IOPS}$$

$$\text{Maximum IOPS per 1000 Mbit Ethernet link} = \frac{1000\text{ Mbit}}{50\text{K} \times 8} = 2500\text{ IOPS}$$

Clearly, the performance bottleneck of the GB2000 is the disks. The IOPS for the whole rack is 40 × 308 or 12,320 IOPS to 40 × 292 or 11,680 IOPS. The network switch would be the bottleneck if it couldn't support 12,320 × 16K × 8 or 1.6 Gbits/sec for 16 KB blocks and 11,680 × 50K × 8 or 4.7 Gbits/sec for 50 KB blocks. We assume that the extra 8 Gbit ports of the 48-port switch connects the rack to the rest of the world, so it could support the full IOPS of the collective 160 disks in the rack.

Using these assumptions, the cost is 40 × ($500 + 4 × $375) + $3000 + $1500 or $84,500 for an 80 TB rack. The disks themselves are almost 60% of the cost. The cost per terabyte is almost $1000, which is about a factor of 10–15 better than storage cluster from the prior edition in 2001. The cost per IOPS is about $7.

Calculating MTTF of the TB-80 Cluster

Internet services like Google rely on many copies of the data at the application level to provide dependability, often at different geographic sites to protect against environmental faults as well as hardware faults. Hence, the Internet Archive has two copies of the data in each site and has sites in San Francisco, Amsterdam, and Alexandria, Egypt. Each site maintains a duplicate copy of the high-value content—music, books, film, and video—and a single copy of the historical Web crawls. To keep costs low, there is no redundancy in the 80 TB rack.

Example Let's look at the resulting mean time to fail of the rack. Rather than use the manufacturer's quoted MTTF of 600,000 hours, we'll use data from a recent survey of disk drives [Gray and van Ingen 2005]. As mentioned in Chapter 1, about 3% to 7% of ATA drives fail per year, or an MTTF of about 125,000–300,000 hours. Make the following assumptions, again assuming exponential lifetimes:

- CPU/memory/enclosure MTTF is 1,000,000 hours.
- PATA Disk MTTF is 125,000 hours.
- PATA controller MTTF is 500,000 hours.
- Ethernet Switch MTTF is 500,000 hours.
- Power supply MTTF is 200,000 hours.
- Fan MTTF is 200,000 hours.
- PATA cable MTTF is 1,000,000 hours.

Answer Collecting these together, we compute these failure rates:

$$\text{Failure rate} = \frac{40}{1{,}000{,}000} + \frac{160}{125{,}000} + \frac{40}{500{,}000} + \frac{1}{500{,}000} + \frac{40}{200{,}000} + \frac{40}{200{,}000} + \frac{80}{1{,}000{,}000}$$

$$= \frac{40 + 1280 + 80 + 2 + 200 + 200 + 80}{1{,}000{,}000 \text{ hours}} = \frac{1882}{1{,}000{,}000 \text{ hours}}$$

The MTTF for the system is just the inverse of the failure rate:

$$\text{MTTF} = \frac{1}{\text{Failure rate}} = \frac{1{,}000{,}000 \text{ hours}}{1882} = 531 \text{ hours}$$

That is, given these assumptions about the MTTF of components, something in a rack fails on average every 3 weeks. About 70% of the failures would be the disks, and about 20% would be fans or power supplies.

6.8 Putting It All Together: NetApp FAS6000 Filer

Network Appliance entered the storage market in 1992 with a goal of providing an easy-to-operate file server running NSF using their own log-structured file system and a RAID 4 disk array. The company later added support for the Windows CIFS file system and a RAID 6 scheme called row-diagonal parity or RAID-DP (see page 364). To support applications that want access to raw data blocks without the overhead of a file system, such as database systems, NetApp filers can serve data blocks over a standard Fibre Channel interface. NetApp also supports *iSCSI*, which allows SCSI commands to run over a TCP/IP network, thereby allowing the use of standard networking gear to connect servers to storage, such as Ethernet, and hence greater distance.

The latest hardware product is the FAS6000. It is a multiprocessor based on the AMD Opteron microprocessor connected using its Hypertransport links. The microprocessors run the NetApp software stack, including NSF, CIFS, RAID-DP, SCSI, and so on. The FAS6000 comes as either a dual processor (FAS6030) or a quad processor (FAS6070). As mentioned in Chapter 4, DRAM is distributed to each microprocessor in the Opteron. The FAS6000 connects 8 GB of DDR2700 to each Opteron, yielding 16 GB for the FAS6030 and 32 GB for the FAS6070. As mentioned in Chapter 5, the DRAM bus is 128 bits wide, plus extra bits for SEC/DED memory. Both models dedicate four Hypertransport links to I/O.

As a filer, the FAS6000 needs a lot of I/O to connect to the disks and to connect to the servers. The integrated I/O consists of

- 8 Fibre Channel (FC) controllers and ports,
- 6 Gigabit Ethernet links,
- 6 slots for x8 (2 GB/sec) PCI Express cards,
- 3 slots for PCI-X 133 MHz, 64-bit cards,
- plus standard I/O options like IDE, USB, and 32-bit PCI.

The 8 Fibre Channel controllers can each be attached to 6 shelves containing 14 3.5-inch FC disks. Thus, the maximum number of drives for the integrated I/O is 8 × 6 × 14 or 672 disks. Additional FC controllers can be added to the option slots to connect up to 1008 drives, to reduce the number of drives per FC network so as to reduce contention, and so on. At 500 GB per FC drive in 2006, if we assume the RAID RDP group is 14 data disks and 2 check disks, the available data capacity is 294 TB for 672 disks and 441 TB for 1008 disks.

It can also connect to Serial ATA disks via a Fibre Channel to SATA bridge controller, which, as its name suggests, allows FC and SATA to communicate.

The six 1-gigabit Ethernet links connect to servers to make the FAS6000 look like a file server running if NTFS or CIFS, or like a block server if running iSCSI.

For greater dependability, FAS6000 filers can be paired so that if one fails, the other can take over. Clustered failover requires that both filers have access to all disks in the pair of filers using the FC interconnect. This interconnect also allows each filer to have a copy of the log data in the NVRAM of the other filer and to keep the clocks of the pair synchronized. The health of the filers is constantly monitored, and failover happens automatically. The healthy filer maintains its own network identity and its own primary functions, but it also assumes the network identity of the failed filer and handles all its data requests via a virtual filer until an administrator restores the data service to the original state.

6.9 Fallacies and Pitfalls

Fallacy *Components fail fast.*

A good deal of the fault-tolerant literature is based on the simplifying assumption that a component operates perfectly until a latent error becomes effective, and then a failure occurs that stops the component.

The Tertiary Disk project had the opposite experience. Many components started acting strangely long before they failed, and it was generally up to the system operator to determine whether to declare a component as failed. The component would generally be willing to continue to act in violation of the service agreement until an operator "terminated" that component.

Figure 6.20 shows the history of four drives that were terminated, and the number of hours they started acting strangely before they were replaced.

Fallacy *Computers systems achieve 99.999% availability ("five nines"), as advertised.*

Marketing departments of companies making servers started bragging about the availability of their computer hardware; in terms of Figure 6.21, they claim availability of 99.999%, nicknamed *five nines*. Even the marketing departments of operating system companies tried to give this impression.

Five minutes of unavailability per year is certainly impressive, but given the failure data collected in surveys, it's hard to believe. For example, Hewlett-Packard claims that the HP-9000 server hardware and HP-UX operating system can deliver a 99.999% availability guarantee "in certain pre-defined, pre-tested customer environments" (see Hewlett-Packard [1998]). This guarantee does not include failures due to operator faults, application faults, or environmental faults,

Messages in system log for failed disk	Number of log messages	Duration (hours)
Hardware Failure (Peripheral device write fault [for] Field Replaceable Unit)	1763	186
Not Ready (Diagnostic failure: ASCQ = Component ID [of] Field Replaceable Unit)	1460	90
Recovered Error (Failure Prediction Threshold Exceeded [for] Field Replaceable Unit)	1313	5
Recovered Error (Failure Prediction Threshold Exceeded [for] Field Replaceable Unit)	431	17

Figure 6.20 Record in system log for 4 of the 368 disks in Tertiary Disk that were replaced over 18 months. See Talagala and Patterson [1999]. These messages, matching the SCSI specification, were placed into the system log by device drivers. Messages started occurring as much as a week before one drive was replaced by the operator. The third and fourth messages indicate that the drive's failure prediction mechanism detected and predicted imminent failure, yet it was still hours before the drives were replaced by the operator.

Unavailability (minutes per year)	Availability (percent)	Availability class ("number of nines")
50,000	90%	1
5,000	99%	2
500	99.9%	3
50	99.99%	4
5	99.999%	5
0.5	99.9999%	6
0.05	99.99999%	7

Figure 6.21 Minutes unavailable per year to achieve availability class (from Gray and Siewiorek [1991]). Note that five nines mean unavailable five minutes per year.

which are likely the dominant fault categories today. Nor does it include scheduled downtime. It is also unclear what the financial penalty is to a company if a system does not match its guarantee.

Microsoft also promulgated a five nines marketing campaign. In January 2001, *www.microsoft.com* was unavailable for 22 hours. For its Web site to achieve 99.999% availability, it will require a clean slate for 250 years.

In contrast to marketing suggestions, well-managed servers in 2006 typically achieve 99% to 99.9% availability.

Pitfall *Where a function is implemented affects its reliability.*

In theory, it is fine to move the RAID function into software. In practice, it is very difficult to make it work reliably.

The software culture is generally based on eventual correctness via a series of releases and patches. It is also difficult to isolate from other layers of software. For example, proper software behavior is often based on having the proper version and patch release of the operating system. Thus, many customers have lost data due to software bugs or incompatibilities in environment in software RAID systems.

Obviously, hardware systems are not immune to bugs, but the hardware culture tends to place a greater emphasis on testing correctness in the initial release. In addition, the hardware is more likely to be independent of the version of the operating system.

Fallacy *Operating systems are the best place to schedule disk accesses.*

Higher-level interfaces like ATA and SCSI offer logical block addresses to the host operating system. Given this high-level abstraction, the best an OS can do is to try to sort the logical block addresses into increasing order. Since only the disk knows the mapping of the logical addresses onto the physical geometry of sectors, tracks, and surfaces, it can reduce the rotational and seek latencies.

For example, suppose the workload is four reads [Anderson 2003]:

Operation	**Starting LBA**	**Length**
Read	724	8
Read	100	16
Read	9987	1
Read	26	128

The host might reorder the four reads into logical block order:

Read	26	128
Read	100	16
Read	724	8
Read	9987	1

Depending on the relative location of the data on the disk, reordering could make it worse, as Figure 6.22 shows. The disk-scheduled reads complete in three-quarters of a disk revolution, but the OS-scheduled reads take three revolutions.

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 from 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 track and assuming there are a large number of tracks. In

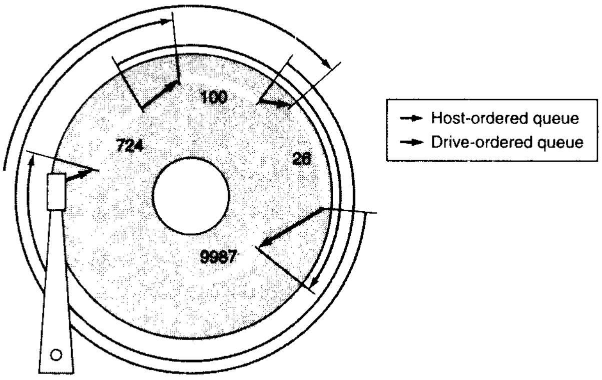


Figure 6.22 Example showing OS versus disk schedule accesses, labeled host-ordered versus drive-ordered. The former takes 3 revolutions to complete the 4 reads, while the latter completes them in just 3/4 of a revolution. From Anderson [2003].

the past, manufacturers listed the seek of this distance to offer a consistent basis for comparison. (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

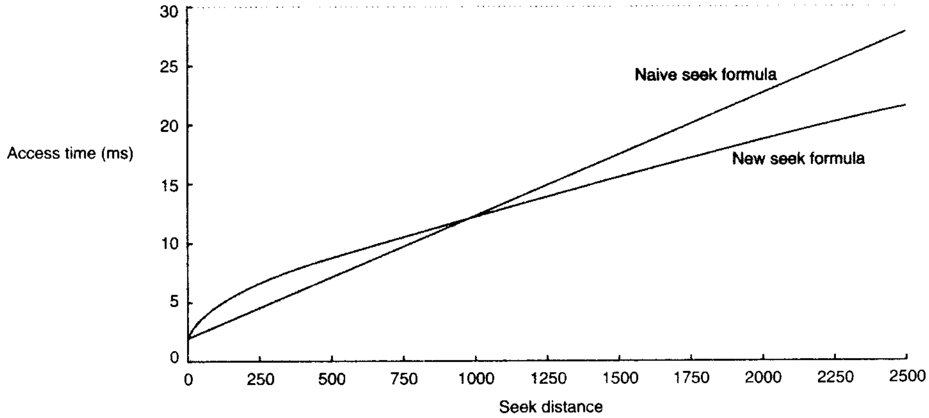
$$\text{Time}_{\text{seek}} = \text{Time}_{\text{minimum}} + \frac{\text{Distance}}{\text{Distance}_{\text{average}}} \times (\text{Time}_{\text{average}} - \text{Time}_{\text{minimum}})$$

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. For disks with more than 200 cylinders, Chen and Lee [1995] modeled the seek distance as

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

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 6.23 plots this equation versus the fallacy equation. Unlike the first equation, the square root of the distance reflects acceleration and deceleration.

The second problem is that the average in the product specification would only be true if there were no locality to disk activity. Fortunately, there is both



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

Figure 6.23 Seek time versus seek distance for sophisticated model versus naive model. Chen and Lee [1995] found that the equations shown above for parameters *a*, *b*, and *c* worked well for several disks.

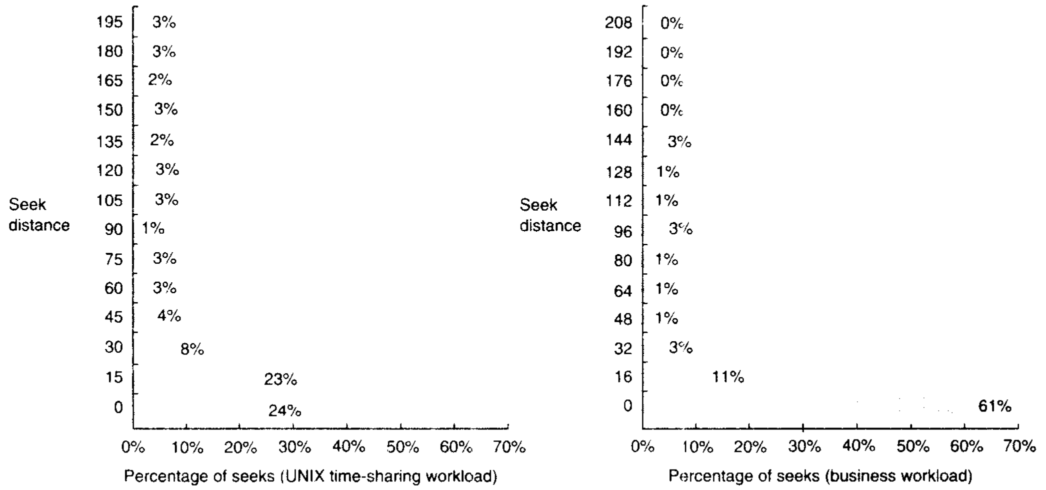


Figure 6.24 Sample measurements of seek distances for two systems. The measurements on the left were taken on a UNIX time-sharing system. The measurements on the right were taken from a business-processing application in which the disk seek activity was scheduled to improve throughput. Seek distance 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 of the 1000 cylinders, but this captured 85% of the accesses. 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 Seagate.

temporal and spatial locality (see page C-2 in Appendix C). For example, Figure 6.24 shows sample measurements of seek distances for two workloads: a UNIX time-sharing 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.

6.10 Concluding Remarks

Storage is one of those technologies that we tend to take for granted. And yet, if we look at the true status of things today, storage is king. One can even argue that servers, which have become commodities, are now becoming peripheral to storage devices. Driving that point home are some estimates from IBM, which expects storage sales to surpass server sales in the next two years.

Michael Vizard
editor in chief, *Infoworld*, August 11, 2001

As their value is becoming increasingly evident, storage systems have become the target of innovation and investment.

The challenge for storage systems today is dependability and maintainability. Not only do users want to be sure their data are never lost (reliability), applications today increasingly demand that the data are always available to access (availability). Despite improvements in hardware and software reliability and fault tolerance, the awkwardness of maintaining such systems is a problem both for cost and for availability. A widely mentioned statistic is that customers spend \$6 to \$8 operating a storage system for every \$1 of purchase price. When dependability is attacked by having many redundant copies at a higher level of the system—such as for search—then very large systems can be sensitive to the price-performance of the storage components.

Today, challenges in storage dependability and maintainability dominate the challenges of I/O.

6.11 Historical Perspective and References

Section K.7 on the companion CD covers the development of storage devices and techniques, including who invented disks, the story behind RAID, and the history of operating systems and databases. References for further reading are included.

Case Studies with Exercises by Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau

Case Study 1: Deconstructing a Disk

Concepts illustrated by this case study

- Performance Characteristics
- Microbenchmarks

The internals of a storage system tend to be hidden behind a simple interface, that of a linear array of blocks. There are many advantages to having a common interface for all storage systems: an operating system can use any storage system without modification, and yet the storage system is free to innovate behind this interface. For example, a single disk can map its internal <sector, track, surface> geometry to the linear array in whatever way achieves the best performance; similarly, a multidisk RAID system can map the blocks on any number of disks to this same linear array. However, this fixed interface has a number of disadvantages as well; in particular, the operating system is not able to perform some performance, reliability, and security optimizations without knowing the precise layout of its blocks inside the underlying storage system.

In this case study, we will explore how software can be used to uncover the internal structure of a storage system hidden behind a block-based interface. The basic idea is to *fingerprint* the storage system: by running a well-defined workload on top of the storage system and measuring the amount of time required for different requests, one is able to infer a surprising amount of detail about the underlying system.

The Skippy algorithm, from work by Nisha Talagala and colleagues at U.C. Berkeley, uncovers the parameters of a single disk. The key is to factor out disk rotational effects by making consecutive seeks to individual sectors with addresses that differ by a linearly increasing amount (increasing by 1, 2, 3, and so forth). Thus, the basic algorithm skips through the disk, increasing the distance of the seek by one sector before every write, and outputs the distance and time for each write. The raw device interface is used to avoid file system optimizations. The `SECTOR SIZE` is set equal to the minimum amount of data that can be read at once from the disk (e.g., 512 bytes). (Skippy is described in more detail in Talagala et al. [1999].)

```
fd = open("raw disk device");
for (i = 0; i < measurements; i++) {
    begin_time = gettime();
    lseek(fd, i*SECTOR_SIZE, SEEK_CUR);
    write(fd, buffer, SECTOR_SIZE);
    interval_time = gettime() -begin_time;

    printf("Stride: %d Time: %d\n", i, interval_time);
}
close(fd);
```

By graphing the time required for each write as a function of the seek distance, one can infer the minimal transfer time (with no seek or rotational latency), head switch time, cylinder switch time, rotational latency, and the number of heads in the disk. A typical graph will have four distinct lines, each with the same slope, but with different offsets. The highest and lowest lines correspond to requests that incur different amounts of rotational delay, but no cylinder or head switch costs: the difference between these two lines reveals the rotational latency of the disk. The second lowest line corresponds to requests that incur a head switch (in addition to increasing amounts of rotational delay). Finally, the third line corresponds to requests that incur a cylinder switch (in addition to rotational delay).

6.1 [10/10/10/10/10] <6.2> The results of running Skippy are shown for a mock disk (Disk Alpha) in Figure 6.25.

a. [10] <6.2> What is the minimal transfer time?

b. [10] <6.2> What is the rotational latency?

c. [10] <6.2> What is the head switch time?

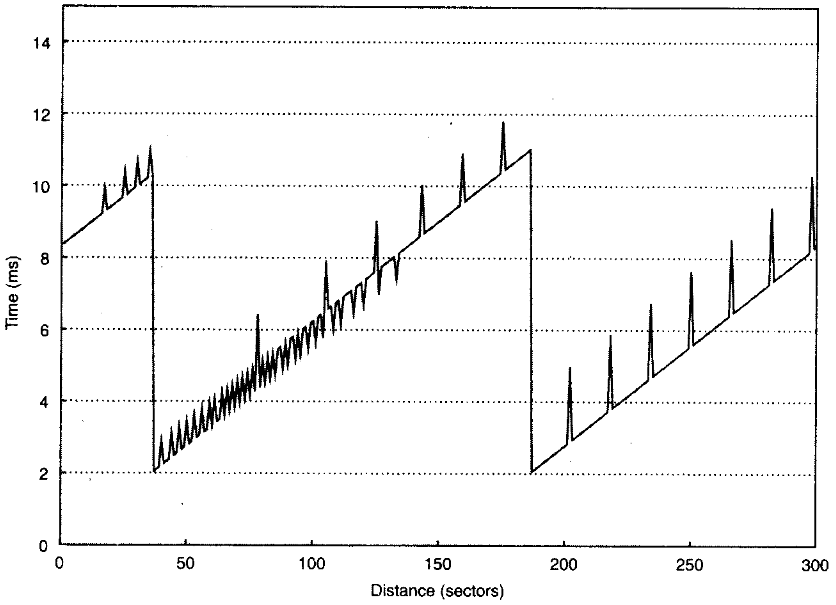


Figure 6.25 Results from running Skippy on Disk Alpha.

d. [10] <6.2> What is the cylinder switch time?

e. [10] <6.2> What is the number of disk heads?

6.2 [25] <6.2> Draw an approximation of the graph that would result from running Skippy on Disk Beta, a disk with the following parameters:

- Minimal transfer time: 2.0 ms
- Rotational latency: 6.0 ms
- Head switch time: 1.0 ms
- Cylinder switch time: 1.5 ms
- Number of disk heads: 4
- Sectors per track: 100

6.3 [10/10/10/10/10/10/10] <6.2> Implement and run the Skippy algorithm on a disk drive of your choosing.

a. [10] <6.2> Graph the results of running Skippy. Report the manufacturer and model of your disk.

b. [10] <6.2> What is the minimal transfer time?

c. [10] <6.2> What is the rotational latency?

d. [10] <6.2> What is the head switch time?

e. [10] <6.2> What is the cylinder switch time?

f. [10] <6.2> What is the number of disk heads?

g. [10] <6.2> Do the results of running Skippy on a real disk differ in any qualitative way from that of the mock disk?

Case Study 2: Deconstructing a Disk Array

Concepts illustrated by this case study

- Performance Characteristics
- Microbenchmarks

The Shear algorithm, from work by Timothy Denehy and colleagues at the University of Wisconsin [Denehy et al. 2004], uncovers the parameters of a RAID system. The basic idea is to generate a workload of requests to the RAID array and time those requests; by observing which sets of requests take longer, one can infer which blocks are allocated to the same disk.

We define RAID properties as follows. Data is allocated to disks in the RAID at the block level, where a *block* is the minimal unit of data that the file system reads or writes from the storage system; thus, block size is known by the file system and the fingerprinting software. A *chunk* is a set of blocks that is allocated contiguously within a disk. A *stripe* is a set of chunks across each of D data disks. Finally, a *pattern* is the minimum sequence of data blocks such that block offset i within the pattern is always located on disk j.

6.4 [20/20] <6.2> One can uncover the pattern size with the following code. The code accesses the raw device to avoid file system optimizations. The key to all of the Shear algorithms is to use random requests to avoid triggering any of the prefetch or caching mechanisms within the RAID or within individual disks. The basic idea of this code sequence is to access N random blocks at a fixed interval p within the RAID array and to measure the completion time of each interval.

```
for (p = BLOCKSIZE; p <= testsize; p += BLOCKSIZE) {
    for (i = 0; i < N; i++) {
        request[i] = random()*p;
    }
    begin_time = gettime();
    issues all request[N] to raw device in parallel;
    wait for all request[N] to complete;
    interval_time = gettime() - begin_time;
    printf("PatternSize: %d Time: %d\n", p,
        interval_time);
}
```

If you run this code on a RAID array and plot the measured time for the N requests as a function of p, then you will see that the time is highest when all N

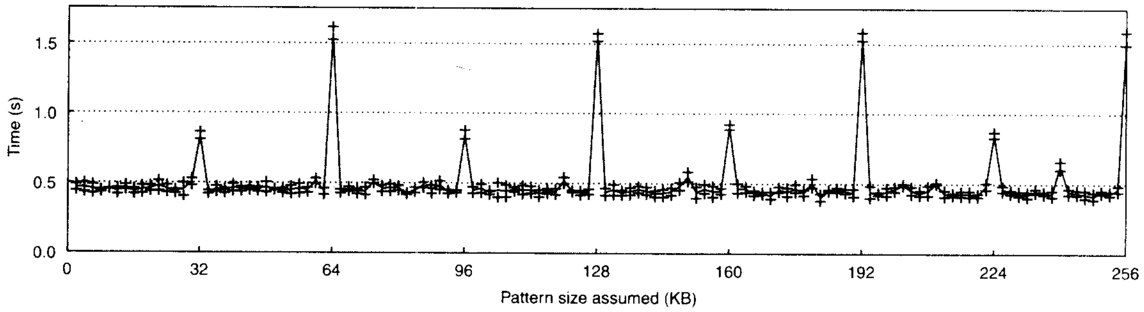


Figure 6.26 Results from running the pattern size algorithm of Shear on a mock storage system.

requests fall on the same disk; thus, the value of p with the highest time corresponds to the pattern size of the RAID.

a. [20] <6.2> Figure 6.26 shows the results of running the pattern size algorithm on an unknown RAID system.

- What is the pattern size of this storage system?
- What do the measured times of 0.4, 0.8, and 1.6 seconds correspond to in this storage system?
- If this is a RAID 0 array, then how many disks are present?
- If this is a RAID 0 array, then what is the chunk size?

b. [20] <6.2> Draw the graph that would result from running this Shear code on a storage system with the following characteristics:

- Number of requests: $N = 1000$
- Time for a random read on disk: 5 ms
- RAID level: RAID 0
- Number of disks: 4
- Chunk size: 8 KB

6.5 [20/20] <6.2> One can uncover the chunk size with the following code. The basic idea is to perform reads from N patterns chosen at random, but always at controlled offsets, c and $c - 1$, within the pattern.

```
for (c = 0; c < patternsize; c += BLOCKSIZE) {
    for (i = 0; i < N; i++) {
        requestA[i] = random()*patternsize + c;
        requestB[i] = random()*patternsize +
            (c-1)%patternsize;
    }
```

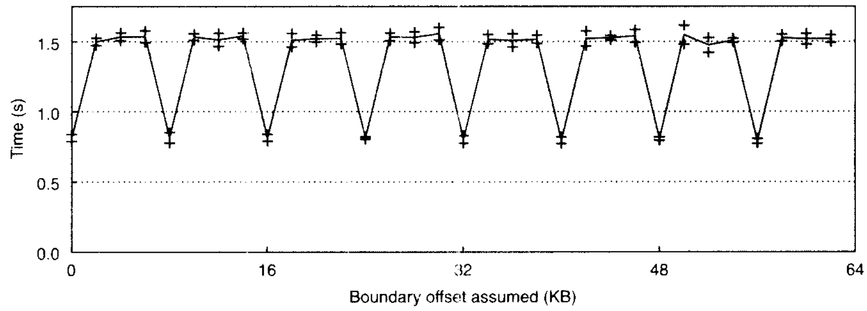


Figure 6.27 Results from running the chunk size algorithm of Shear on a mock storage system.

```
        begin_time = gettime();

        issue all requestA[N] and requestB[N] to raw device
            in parallel;
        wait for requestA[N] and requestB[N] to complete;

        interval_time = gettime() - begin_time;
        printf("ChunkSize: %d Time: %d\n", c, interval_time);
    }
```

If you run this code and plot the measured time as a function of *c*, then you will see that the measured time is lowest when the *requestA* and *requestB* reads fall on two different disks. Thus, the values of *c* with low times correspond to the chunk boundaries between disks of the RAID.

a. [20] <6.2> Figure 6.27 shows the results of running the chunk size algorithm on an unknown RAID system.

- What is the chunk size of this storage system?
- What do the measured times of 0.75 and 1.5 seconds correspond to in this storage system?

b. [20] <6.2> Draw the graph that would result from running this Shear code on a storage system with the following characteristics:

- Number of requests: $N = 1000$
- Time for a random read on disk: 5 ms
- RAID level: RAID 0
- Number of disks: 8
- Chunk size: 12 KB

6.6 [10/10/10/10] <6.2> Finally, one can determine the layout of chunks to disks with the following code. The basic idea is to select N random patterns, and to exhaustively read together all pairwise combinations of the chunks within the pattern.

```
for (a = 0; a < numchunks; a += chunksize) {
    for (b = a; b < numchunks; b += chunksize) {
        for (i = 0; i < N; i++) {
            requestA[i] = random()*patternsize + a;
            requestB[i] = random()*patternsize + b;
        }
        begin_time = gettime();
        issue all requestA[N] and requestB[N] to raw device
        in parallel;
        wait for all requestA[N] and requestB[N] to
            complete;
        interval_time = gettime() - begin_time;
        printf("A: %d B: %d Time: %d\n", a, b,
            interval_time);
    }
}
```

After running this code, you can report the measured time as a function of *a* and *b*. The simplest way to graph this is to create a two-dimensional table with *a* and *b* as the parameters, and the time scaled to a shaded value: we use darker shadings for faster times and lighter shadings for slower times. Thus, a light shading indicates that the two offsets of *a* and *b* within the pattern fall on the same disk.

Figure 6.28 shows the results of running the layout algorithm on a storage system that is known to have a pattern size of 384 KB and a chunk size of 32 KB.

a. [20] <6.2> How many chunks are in a pattern?

b. [20] <6.2> Which chunks of each pattern appear to be allocated on the same disks?

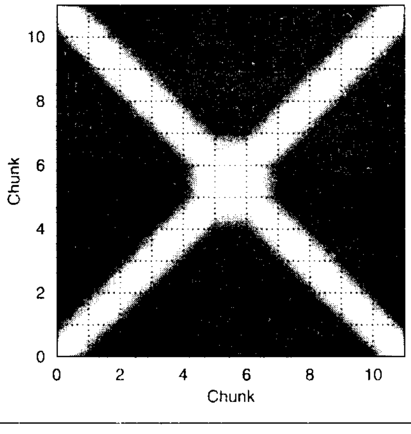
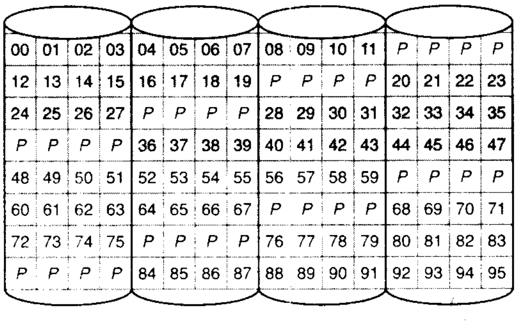


Figure 6.28 Results from running the layout algorithm of Shear on a mock storage system.



00	01	02	03	04	05	06	07	08	09	10	11	P	P	P	P
12	13	14	15	16	17	18	19	P	P	P	P	20	21	22	23
24	25	26	27	P	P	P	P	28	29	30	31	32	33	34	35
P	P	P	P	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	P	P	P	P
60	61	62	63	64	65	66	67	P	P	P	P	68	69	70	71
72	73	74	75	P	P	P	P	76	77	78	79	80	81	82	83
P	P	P	P	84	85	86	87	88	89	90	91	92	93	94	95

Parity: RAID 5 Left-Asymmetric, stripe = 16, pattern = 48

Figure 6.29 A storage system with 4 disks, a chunk size of four 4 KB blocks, and using a RAID 5 Left-Asymmetric layout. Two repetitions of the pattern are shown.

c. [20] <6.2> How many disks appear to be in this storage system?

d. [20] <6.2> Draw the likely layout of blocks across the disks.

6.7 [20] <6.2> Draw the graph that would result from running the layout algorithm on the storage system shown in Figure 6.29. This storage system has 4 disks, a chunk size of four 4 KB blocks (16 KB), and is using a RAID 5 Left-Asymmetric layout.

Case Study 3: RAID Reconstruction

Concepts illustrated by this case study

- RAID Systems
- RAID Reconstruction
- Mean Time to Failure (MTTF)
- Mean Time until Data Loss (MTDL)
- Performability
- Double Failures

A RAID system ensures that data is not lost when a disk fails. Thus, one of the key responsibilities of a RAID is to reconstruct the data that was on a disk when it failed; this process is called *reconstruction* and is what you will explore in this case study. You will consider both a RAID system that can tolerate one disk failure, and a RAID-DP, which can tolerate two disk failures.

Reconstruction is commonly performed in two different ways. In *off-line reconstruction*, the RAID devotes all of its resources to performing reconstruction and does not service any requests from the workload. In *on-line reconstruction*, the RAID continues to service workload requests while performing the reconstruction; the reconstruction process is often limited to use some fraction of the total bandwidth of the RAID system.

How reconstruction is performed impacts both the *reliability* and the *performability* of the system. In a RAID 5, data is lost if a second disk fails before the data from the first disk is recovered; therefore, the longer the reconstruction time (MTTR), the lower the reliability or the *mean time until data loss* (MTDL). Performability is a metric meant to combine both the performance of a system and its availability; it is defined as the performance of the system in a given state multiplied by the probability of that state. For a RAID array, possible states include normal operation with no disk failures, reconstruction with one disk failure, and shutdown due to multiple disk failures.

For these exercises, assume that you have built a RAID system with six disks, plus a sufficient number of hot spares. Assume each disk is the 37 GB SCSI disk shown in Figure 6.3; assume each disk can sequentially read data at a peak of 142 MB/sec and sequentially write data at a peak of 85 MB/sec. Assume that the disks are connected to an Ultra320 SCSI bus that can transfer a total of 320 MB/sec. You can assume that each disk failure is independent and ignore other potential failures in the system. For the reconstruction process, you can assume that the overhead for any XOR computation or memory copying is negligible. During online reconstruction, assume that the reconstruction process is limited to use a total bandwidth of 10 MB/sec from the RAID system.

6.8 [10] <6.2> Assume that you have a RAID 4 system with six disks. Draw a simple diagram showing the layout of blocks across disks for this RAID system.

6.9 [10] <6.2, 6.4> When a single disk fails, the RAID 4 system will perform reconstruction. What is the expected time until a reconstruction is needed?

6.10 [10/10/10] <6.2, 6.4> Assume that reconstruction of the RAID 4 array begins at time *t*.

a. [10] <6.2, 6.4> What read and write operations are required to perform the reconstruction?

b. [10] <6.2, 6.4> For offline reconstruction, when will the reconstruction process be complete?

c. [10] <6.2, 6.4> For online reconstruction, when will the reconstruction process be complete?

6.11 [10/10/10/10] <6.2, 6.4> In this exercise, we will investigate the mean time until data loss (MTDL). In RAID 4, data is lost only if a second disk fails before the first failed disk is repaired.

a. [10] <6.2, 6.4> What is the likelihood of having a second failure during offline reconstruction?

b. [10] <6.2, 6.4> Given this likelihood of a second failure during reconstruction, what is the MTDL for offline reconstruction?

c. [10] <6.2, 6.4> What is the likelihood of having a second failure during online reconstruction?

d. [10] <6.2, 6.4> Given this likelihood of a second failure during reconstruction, what is the MTDL for online reconstruction?

6.12 [10] <6.2, 6.4> What is performability for the RAID 4 array for offline reconstruction? Calculate the performability using IOPS, assuming a random read-only workload that is evenly distributed across the disks of the RAID 4 array.

6.13 [10] <6.2, 6.4> What is the performability for the RAID 4 array for online reconstruction? During online repair, you can assume that the IOPS drop to 70% of their peak rate. Does offline or online reconstruction lead to better performability?

6.14 [10] <6.2, 6.4> RAID 6 is used to tolerate up to two simultaneous disk failures. Assume that you have a RAID 6 system based on row-diagonal parity, or RAID-DP; your six-disk RAID-DP system is based on RAID 4, with $p = 5$, as shown in Figure 6.5. If data disk 0 and data disk 3 fail, how can those disks be reconstructed? Show the sequence of steps that are required to compute the missing blocks in the first four stripes.

Case Study 4: Performance Prediction for RAIDs

Concepts illustrated by this case study

- RAID Levels
- Queuing Theory
- Impact of Workloads
- Impact of Disk Layout

In this case study, you will explore how simple queuing theory can be used to predict the performance of the I/O system. You will investigate how both storage system configuration and the workload influence service time, disk utilization, and average response time.

The configuration of the storage system has a large impact on performance. Different RAID levels can be modeled using queuing theory in different ways. For example, a RAID 0 array containing N disks can be modeled as N separate systems of M/M/1 queues, assuming that requests are appropriately distributed across the N disks. The behavior of a RAID 1 array depends upon the workload: a read operation can be sent to either mirror, whereas a write operation must be sent to both disks. Therefore, for a read-only workload, a two-disk RAID 1 array can be modeled as an M/M/2 queue, whereas for a write-only workload, it can be modeled as an M/M/1 queue. The behavior of a RAID 4 array containing N disks also depends upon the workload: a read will be sent to a particular data disk, whereas writes must all update the parity disk, which becomes the bottleneck of the system. Therefore, for a read-only workload, RAID 4 can be modeled as $N - 1$ separate systems, whereas for a write-only workload, it can be modeled as one M/M/1 queue.

The layout of blocks within the storage system can have a significant impact on performance. Consider a single disk with a 40 GB capacity. If the workload randomly accesses 40 GB of data, then the layout of those blocks to the disk does not have much of an impact on performance. However, if the workload randomly accesses only half of the disk's capacity (i.e., 20 GB of data on that disk), then layout does matter: to reduce seek time, the 20 GB of data can be compacted within 20 GB of consecutive tracks instead of allocated uniformly distributed over the entire 40 GB capacity.

For this problem, we will use a rather simplistic model to estimate the service time of a disk. In this basic model, the average positioning and transfer time for a small random request is a linear function of the seek distance. For the 40 GB disk in this problem, assume that the service time is 5 *ms* * space utilization. Thus, if the entire 40 GB disk is used, then the average positioning and transfer time for a random request is 5 ms; if only the first 20 GB of the disk is used, then the average positioning and transfer time is 2.5 ms.

Throughout this case study, you can assume that the processor sends 167 small random disk requests per second and that these requests are exponentially distributed. You can assume that the size of the requests is equal to the block size of 8 KB. Each disk in the system has a capacity of 40 GB. Regardless of the storage system configuration, the workload accesses a total of 40 GB of data; you should allocate the 40 GB of data across the disks in the system in the most efficient manner.

6.15 [10/10/10/10/10] <6.5> Begin by assuming that the storage system consists of a single 40 GB disk.

a. [10] <6.5> Given this workload and storage system, what is the average service time?

b. [10] <6.5> On average, what is the utilization of the disk?

c. [10] <6.5> On average, how much time does each request spend waiting for the disk?

d. [10] <6.5> What is the mean number of requests in the queue?

e. [10] <6.5> Finally, what is the average response time for the disk requests?

6.16 [10/10/10/10/10/10] <6.2, 6.5> Imagine that the storage system is now configured to contain two 40 GB disks in a RAID 0 array; that is, the data is striped in blocks of 8 KB equally across the two disks with no redundancy.

a. [10] <6.2, 6.5> How will the 40 GB of data be allocated across the disks? Given a random request workload over a total of 40 GB, what is the expected service time of each request?

b. [10] <6.2, 6.5> How can queuing theory be used to model this storage system?

c. [10] <6.2, 6.5> What is the average utilization of each disk?

d. [10] <6.2, 6.5> On average, how much time does each request spend waiting for the disk?

e. [10] <6.2, 6.5> What is the mean number of requests in each queue?

f. [10] <6.2, 6.5> Finally, what is the average response time for the disk requests?

6.17 [20/20/20/20/20] <6.2, 6.5> Instead imagine that the storage system is configured to contain two 40 GB disks in a RAID 1 array; that is, the data is mirrored across the two disks. Use queuing theory to model this system for a read-only workload.

a. [20] <6.2, 6.5> How will the 40 GB of data be allocated across the disks? Given a random request workload over a total of 40 GB, what is the expected service time of each request?

b. [20] <6.2, 6.5> How can queuing theory be used to model this storage system?

c. [20] <6.2, 6.5> What is the average utilization of each disk?

d. [20] <6.2, 6.5> On average, how much time does each request spend waiting for the disk?

e. [20] <6.2, 6.5> Finally, what is the average response time for the disk requests?

6.18 [10/10] <6.2, 6.5> Imagine that instead of a read-only workload, you now have a write-only workload on a RAID 1 array.

a. [10] <6.2, 6.5> Describe how you can use queuing theory to model this system and workload.

b. [10] <6.2, 6.5> Given this system and workload, what is the average utilization, average waiting time, and average response time?

Case Study 5: I/O Subsystem Design

Concepts illustrated by this case study

- RAID Systems
- Mean Time to Failure (MTTF)
- Performance and Reliability Trade-offs

In this case study, you will design an I/O subsystem, given a monetary budget. Your system will have a minimum required capacity and you will optimize for performance, reliability, or both. You are free to use as many disks and controllers as fit within your budget.

Here are your building blocks:

- A 10,000 MIPS CPU costing $1000. Its MTTF is 1,000,000 hours.
- A 1000 MB/sec I/O bus with room for 20 Ultra320 SCSI buses and controllers.
- Ultra320 SCSI buses that can transfer 320 MB/sec and support up to 15 disks per bus (these are also called SCSI strings). The SCSI cable MTTF is 1,000,000 hours.

- An Ultra320 SCSI controller that is capable of 50,000 IOPS, costs $250, and has an MTTF of 500,000 hours.
- A $2000 enclosure supplying power and cooling to up to eight disks. The enclosure MTTF is 1,000,000 hours, the fan MTTF is 200,000 hours, and the power supply MTTF is 200,000 hours.
- The SCSI disks described in Figure 6.3.
- Replacing any failed component requires 24 hours.

You may make the following assumptions about your workload:

- The operating system requires 70,000 CPU instructions for each disk I/O.
- The workload consists of many concurrent, random I/Os, with an average size of 16 KB.

All of your constructed systems must have the following properties:

- You have a monetary budget of $28,000.
- You must provide at least 1 TB of capacity.

6.19 [10] <6.2> You will begin by designing an I/O subsystem that is optimized only for capacity and performance (and not reliability), specifically IOPS. Discuss the RAID level and block size that will deliver the best performance.

6.20 [20/20/20/20] <6.2, 6.4, 6.7> What configuration of SCSI disks, controllers, and enclosures results in the best performance given your monetary and capacity constraints?

a. [20] <6.2, 6.4, 6.7> How many IOPS do you expect to deliver with your system?

b. [20] <6.2, 6.4, 6.7> How much does your system cost?

c. [20] <6.2, 6.4, 6.7> What is the capacity of your system?

d. [20] <6.2, 6.4, 6.7> What is the MTTF of your system?

6.21 [10] <6.2, 6.4, 6.7> You will now redesign your system to optimize for reliability, by creating a RAID 10 or RAID 01 array. Your storage system should be robust not only to disk failures, but to controller, cable, power supply, and fan failures as well; specifically, a single component failure should not prohibit accessing both replicas of a pair. Draw a diagram illustrating how blocks are allocated across disks in the RAID 10 and RAID 01 configurations. Is RAID 10 or RAID 01 more appropriate in this environment?

6.22 [20/20/20/20/20] <6.2, 6.4, 6.7> Optimizing your RAID 10 or RAID 01 array only for reliability (but keeping within your capacity and monetary constraints), what is your RAID configuration?

a. [20] <6.2, 6.4, 6.7> What is the overall MTTF of the components in your system?

b. [20] <6.2, 6.4, 6.7> What is the MTDL of your system?

c. [20] <6.2, 6.4, 6.7> What is the usable capacity of this system?

d. [20] <6.2, 6.4, 6.7> How much does your system cost?

e. [20] <6.2, 6.4, 6.7> Assuming a write-only workload, how many IOPS can you expect to deliver?

6.23 [10] <6.2, 6.4, 6.7> Assume that you now have access to a disk that has twice the capacity, for the same price. If you continue to design only for reliability, how would you change the configuration of your storage system? Why?

Case Study 6: Dirty Rotten Bits

Concepts illustrated by this case study

- Partial Disk Failure
- Failure Analysis
- Performance Analysis
- Parity Protection
- Checksumming

You are put in charge of avoiding the problem of "bit rot"—bits or blocks in a file going bad over time. This problem is particularly important in archival scenarios, where data is written once and perhaps accessed many years later; without taking extra measures to protect the data, the bits or blocks of a file may slowly change or become unavailable due to media errors or other I/O faults.

Dealing with bit rot requires two specific components: detection and recovery. To detect bit rot efficiently, one can use checksums over each block of the file in question; a checksum is just a function of some kind that takes a (potentially long) string of data as input and outputs a fixed-size string (the checksum) of the data as output. The property you will exploit is that if the data changes, the computed checksum is very likely to change as well.

Once detected, recovering from bit rot requires some form of redundancy. Examples include mirroring (keeping multiple copies of each block) and parity (some extra redundant information, usually more space efficient than mirroring).

In this case study. you will analyze how effective these techniques are given various scenarios. You will also write code to implement data integrity protection over a set of files.

6.24 [20/20/20] <6.2> Assume that you will use simple parity protection in Exercises 6.24 through 6.27. Specifically, assume that you will be computing *one* parity block for each file in the file system. Further, assume that you will also use a 20-byte MD5 checksum per 4 KB block of each file.

We first tackle the problem of space overhead. According to recent studies [Douceur and Bolosky 1999], these file size distributions are what is found in modern PCs:

≤1 KB	2 KB	4 KB	8 KB	16 KB	32 KB	64 KB	128 KB	256 KB	512 KB	≥1 MB
26.6%	11.0%	11.2%	10.9%	9.5%	8.5%	7.1%	5.1%	3.7%	2.4%	4.0%

The study also finds that file systems are usually about half full. Assume you have a 37 GB disk volume that is roughly half full and follows that same distribution, and answer the following questions:

a. [20] <6.2> How much extra information (both in bytes and as a percent of the volume) must you keep on disk to be able to detect a single error with checksums?

b. [20] <6.2> How much extra information (both in bytes and as a percent of the volume) would you need to be able to both detect a single error with checksums as well as correct it?

c. [20] <6.2> Given this file distribution, is the block size you are using to compute checksums too big, too little, or just right?

6.25 [10/10] <6.2, 6.3> One big problem that arises in data protection is error detection. One approach is to perform error detection *lazily*—that is, wait until a file is accessed, and at that point, check it and make sure the correct data is there. The problem with this approach is that files that are not accessed frequently may thus slowly rot away, and when finally accessed, have too many errors to be corrected. Hence, an eager approach is to perform what is sometimes called *disk scrubbing*—periodically go through all data and find errors proactively.

a. [10] <6.2, 6.3> Assume that bit flips occur independently, at a rate of 1 flip per GB of data per month. Assuming the same 20 GB volume that is half full, and assuming that you are using the SCSI disk as specified in Figure 6.3 (4 ms seek, roughly 100 MB/sec transfer), how often should you scan through files to check and repair their integrity?

b. [10] <6.2, 6.3> At what bit flip rate does it become impossible to maintain data integrity? Again assume the 20 GB volume and the SCSI disk.

6.26 [10/10/10/10] <6.2, 6.4> Another potential cost of added data protection is found in performance overhead. We now study the performance overhead of this data protection approach.

a. [10] <6.2, 6.4> Assume we write a 40 MB file to the SCSI disk sequentially, and then write out the extra information to implement our data protection scheme to disk once. How much *write traffic* (both in total volume of bytes and as a percentage of total traffic) does our scheme generate?

b. [10] <6.2, 6.4> Assume we now are updating the file randomly, similar to a database table. That is, assume we perform a series of 4 KB random writes to the file, and each time we perform a single write, we must update the on-disk protection information. Assuming that we perform 10,000 random writes, how much *I/O traffic* (both in total volume of bytes and as a percentage of total traffic) does our scheme generate?

c. [10] <6.2, 6.4> Now assume that the data protection information is always kept in a separate portion of the disk, away from the file it is guarding (that is, assume for each file A, there is another file $A_{\text{checksums}}$ that holds all the checksums for A). Hence, one potential overhead we must incur arises upon reads—that is, upon each read, we will use the checksum to detect data corruption.

   Assume you read 10,000 blocks of 4 KB each sequentially from disk. Assuming a 4 ms average seek cost and a 100 MB/sec transfer rate (like the SCSI disk in Figure 6.3), how long will it take to read the file (and corresponding checksums) from disk? What is the time penalty due to adding checksums?

d. [10] <6.2, 6.4> Again assuming that the data protection information is kept separate as in part (c), now assume you have to read 10,000 random blocks of 4 KB each from a very large file (much bigger than 10,000 blocks, that is). For each read, you must again use the checksum to ensure data integrity. How long will it take to read the 10,000 blocks from disk, again assuming the same disk characteristics? What is the time penalty due to adding checksums?

6.27 [40] <6.2, 6.3, 6.4> Finally, we put theory into practice by developing a user-level tool to guard against file corruption. Assume you are to write a simple set of tools to detect and repair data integrity. The first tool is used to checksums and parity. It should be called `build` and used like this:

```
build <filename>
```

The `build` program should then store the needed checksum and redundancy information for the file `filename` in a file in the same directory called `.filename.cp` (so it is easy to find later).

A second program is then used to check and potentially repair damaged files. It should be called `repair` and used like this:

```
repair <filename>
```

The `repair` program should consult the `.cp` file for the filename in question and verify that all the stored checksums match the computed checksums for the data. If the checksums don't match for a single block, `repair` should use the redundant information to reconstruct the correct data and fix the file. However, if two or more blocks are bad, `repair` should simply report that the file has been corrupted beyond repair. To test your system, we will provide a tool to corrupt files called `corrupt`. It works as follows:

```
corrupt <filename> <blocknumber>
```

All `corrupt` does is fill the specified block number of the file with random noise. For checksums you will be using MD5. MD5 takes an input string and gives you a 128-bit "fingerprint" or checksum as an output. A great and simple implementation of MD5 is available here:

```
http://sourceforge.net/project/showfiles.php?group_id=42360
```

Parity is computed with the XOR operator. In C code, you can compute the parity of two blocks, each of size `BLOCKSIZE`, as follows:

```
unsigned char block1[BLOCKSIZE];
unsigned char block2[BLOCKSIZE];

unsigned char parity[BLOCKSIZE];

// first, clear parity block
for (int i = 0; i < BLOCKSIZE; i++)
    parity[i] = 0;

// then compute parity; carat symbol does XOR in C
for (int i = 0; i < BLOCKSIZE; i++) {
    parity[i] = block1[i] ^ block2[i];
}
```

Case Study 7: Sorting Things Out

Concepts illustrated by this case study

- Benchmarking
- Performance Analysis
- Cost/Performance Analysis
- Amortization of Overhead
- Balanced Systems

The database field has a long history of using benchmarks to compare systems. In this question, you will explore one of the benchmarks introduced by Anonymous et al. [1985] (see Chapter 1): external, or disk-to-disk, sorting.

Sorting is an exciting benchmark for a number of reasons. First, sorting exercises a computer system across all its components, including disk, memory, and processors. Second, sorting at the highest possible performance requires a great deal of expertise about how the CPU caches, operating systems, and I/O subsystems work. Third, it is simple enough to be implemented by a student (see below!).

Depending on how much data you have, sorting can be done in one or multiple passes Simply put, if you have enough memory to hold the entire data set in memory, you can read the entire data set into memory, sort it, and then write it out; this is called a "one-pass" sort.

If you do not have enough memory, you must sort the data in multiple passes. There are many different approaches possible. One simple approach is to sort each chunk of the input file and write it to disk; this leaves (input file size)/(memory size) sorted files on disk. Then, you have to merge each sorted temporary file into a final sorted output. This is called a "two-pass" sort. More passes are needed in the unlikely case that you cannot merge all the streams in the second pass.

In this case study you will analyze various aspects of sorting, determining its effectiveness and cost-effectiveness in different scenarios. You will also write your own version of an external sort, measuring its performance on real hardware.

6.28 [20/20/20] <6.4> We will start by configuring a system to complete a sort in the least possible time, with no limits on how much we can spend. To get peak bandwidth from the sort, we have to make sure all the paths through the system have sufficient bandwidth.

Assume for simplicity that the time to perform the in-memory sort of keys is linearly proportional to the CPU rate and memory bandwidth of the given machine (e.g., sorting 1 MB of records on a machine with 1 MB/sec of memory bandwidth and a 1 MIPS processor will take 1 second). Assume further that you have carefully written the I/O phases of the sort so as to achieve sequential bandwidth. And of course realize that if you don't have enough memory to hold all of the data at once that sort will take two passes.

One problem you may encounter in performing I/O is that systems often perform extra *memory copies;* for example, when the `read()` system call is invoked, data may first be read from disk into a system buffer, and then subsequently copied into the specified user buffer. Hence, memory bandwidth during I/O can be an issue.

Finally, for simplicity, assume that there is no overlap of reading, sorting, or writing. That is, when you are reading data from disk, that is all you are doing; when sorting, you are just using the CPU and memory bandwidth; when writing, you are just writing data to disk.

Your job in this task is to configure a system to extract peak performance when sorting 1 GB of data (i.e., roughly 10 million 100-byte records). Use the following table to make choices about which machine, memory, I/O interconnect, and disks to buy.

CPU			**I/O interconnect**		
Slow	1 GIPS	$200	Slow	80 MB/sec	$50
Standard	2 GIPS	$1000	Standard	160 MB/sec	$100
Fast	4 GIPS	$2000	Fast	320 MB/sec	$400
Memory			**Disks**		
Slow	512 MB/sec	$100/GB	Slow	30 MB/sec	$70
Standard	1 GB/sec	$200/GB	Standard	60 MB/sec	$120
Fast	2 GB/sec	$500/GB	Fast	110 MB/sec	$300

Note: Assume you are buying a single-processor system, and that you can have up to two I/O interconnects. However, the amount of memory and number of disks is up to you (assume there is no limit on disks per I/O interconnect).

a. [20] <6.4> What is the total cost of your machine? (Break this down by part, including the cost of the CPU, amount of memory, number of disks, and I/O bus.)

b. [20] <6.4> How much time does it take to complete the sort of 1 GB worth of records? (Break this down into time spent doing reads from disk, writes to disk, and time spent sorting.)

c. [20] <6.4> What is the bottleneck in your system?

6.29 [25/25/25] <6.4> We will now examine cost-performance issues in sorting. After all, it is easy to buy a high-performing machine; it is much harder to buy a cost-effective one.

One place where this issue arises is with the PennySort competition (*research.microsoft.com/barc/SortBenchmark/*). PennySort asks that you sort as many records as you can for a single penny. To compute this, you should assume that a system you buy will last for 3 years (94,608,000 seconds), and divide this by the total cost in pennies of the machine. The result is your time budget per penny.

Our task here will be a little simpler. Assume you have a fixed budget of $2000 (or less). What is the fastest sorting machine you can build? Use the same hardware table as in Exercise 6.28 to configure the winning machine.

(*Hint:* You might want to write a little computer program to generate all the possible configurations.)

a. [25] <6.4> What is the total cost of your machine? (Break this down by part, including the cost of the CPU, amount of memory, number of disks, and I/O bus.)

b. [25] <6.4> How does the reading, writing, and sorting time break down with this configuration?

c. [25] <6.4> What is the bottleneck in your system?

6.30 [20/20/20] <6.4, 6.6> Getting good disk performance often requires *amortization of overhead*. The idea is simple: if you must incur an overhead of some kind, do as much useful work as possible after paying the cost, and hence reduce its impact. This idea is quite general and can be applied to many areas of computer systems; with disks, it arises with the seek and rotational costs (overheads) that you must incur before transferring data. You can amortize an expensive seek and rotation by transferring a large amount of data.

In this exercise, we focus on how to amortize seek and rotational costs during the second pass of a two-pass sort. Assume that when the second pass begins, there are N sorted runs on the disk, each of a size that fits within main memory. Our task here is to read in a chunk from each sorted run and merge the results into a final sorted output. Note that a read from one run will incur a seek and rotation, as it is very likely that the last read was from a different run.

a. [20] <6.4, 6.6> Assume that you have a disk that can transfer at 100 MB/sec, with an average seek cost of 7 ms, and a rotational rate of 10,000 RPM. Assume further that every time you read from a run, you read 1 MB of data, and that there are 100 runs each of size 1 GB. Also assume that writes (to the final sorted output) take place in large 1 GB chunks. How long will the merge phase take, assuming I/O is the dominant (i.e., only) cost?

b. [20] <6.4, 6.6> Now assume that you change the read size from 1 MB to 10 MB. How is the total time to perform the second pass of the sort affected?

c. [20] <6.4, 6.6> In both cases, assume that what we wish to maximize is *disk efficiency*. We compute disk efficiency as the ratio of the time spent transferring data over the total time spent accessing the disk. What is the disk efficiency in each of the scenarios mentioned above?

6.31 [40] <6.2, 6.4, 6.6> In this exercise, you will write your own external sort. To generate the data set, we provide a tool `generate` that works as follows:

```
generate <filename> <size (in MB)>
```

By running `generate`, you create a file named `filename` of size `size` MB. The file consists of 100 byte keys, with 10-byte records (the part that must be sorted).

We also provide a tool called `check` that checks whether a given input file is sorted or not. It is run as follows:

```
check <filename>
```

The basic one-pass sort does the following: reads in the data, sorts it, and then writes it out. However, numerous optimizations are available to you: overlapping reading and sorting, separating keys from the rest of the record for better cache behavior and hence faster sorting, overlapping sorting and writing, and so forth. Neuberg et al. [1994] is a terrific place to look for some hints.

One important rule is that data must always start on disk (and not in the file system cache. The easiest way to ensure this is to unmount and remount the file system.

One goal: beat the Datamation sort record. Currently, the record for sorting 1 million 100-byte records is 0.44 seconds, which was obtained on a cluster of 32 machines. If you are careful, you might be able to beat this on a single PC configured with a few disks.