
D.1	Introduction	D-2
D.2	Advanced Topics in Disk Storage	D-2
D.3	Definition and Examples of Real Faults and Failures	D-10
D.4	I/O Performance, Reliability Measures, and Benchmarks	D-15
D.5	A Little Queuing Theory	D-23
D.6	Crosscutting Issues	D-34
D.7	Designing and Evaluating an I/O System—The Internet Archive Cluster	D-36
D.8	Putting It All Together: NetApp FAS6000 Filer	D-41
D.9	Fallacies and Pitfalls	D-43
D.10	Concluding Remarks	D-47
D.11	Historical Perspective and References Case Studies with Exercises by Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau	D-48

D

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)*

D.1 Introduction

The popularity of Internet services such as 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 F.

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 that readers are already familiar with the basics of storage devices, some of which were covered in Chapter 1.

D.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 3 years. Between then and about 1996, the rate improved to 60% per year, quadrupling density every 3 years and matching the traditional rate of DRAMs. From 1997 to about 2003, the rate increased to 100%, doubling every year. After the innovations that allowed this renaissance had largely played out, the rate has dropped recently to about 30% per year. In 2011, the highest density in commercial products is 400 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 almost a factor of 1,000,000 between 1983 and 2011.

Magnetic disks have been challenged many times for supremacy of secondary storage. Figure D.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 to 150 times more per gigabyte for DRAM.

The bandwidth gap is more complex. For example, a fast disk in 2011 transfers at 200 MB/sec from the disk media with 600 GB of storage and costs about \$400. A 4 GB DRAM module costing about \$200 in 2011 could transfer at 16,000 MB/sec (see Chapter 2), giving the DRAM module about 80 times higher bandwidth than the disk. However, the bandwidth per GB is 6000 times higher for DRAM, and the bandwidth per dollar is 160 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 ships, 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 non-volatile like disks, and it has about the same bandwidth as disks, but latency is 100 to 1000 times faster than disk. In 2011, the price per gigabyte of Flash was 15 to 20 times cheaper than DRAM. Flash is popular in cell phones because it comes in much smaller capacities and it is more power efficient than disks, despite the cost per gigabyte being 15 to 25 times higher than disks. Unlike disks and DRAM,

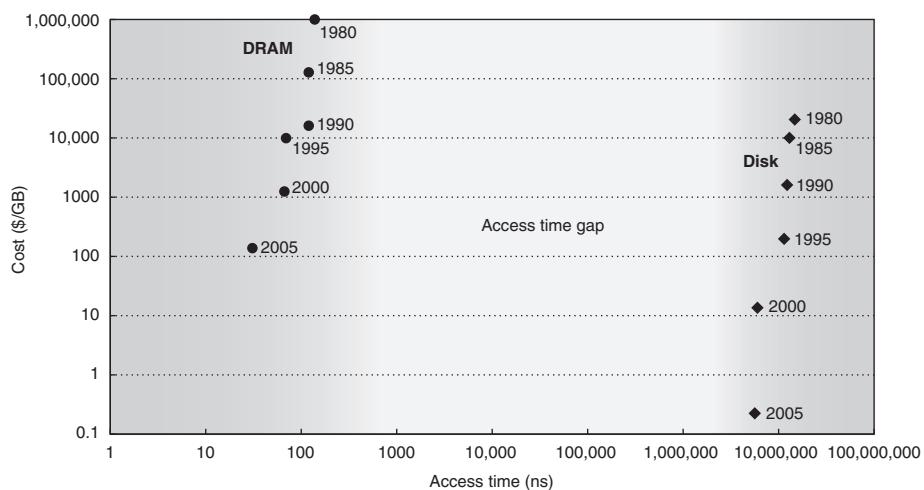


Figure D.1 Cost versus access time for DRAM and magnetic disk in 1980, 1985, 1990, 1995, 2000, and 2005. The two-order-of-magnitude gap in cost and five-order-of-magnitude gap in access times between semiconductor memory and rotating magnetic disks have inspired a host of competing technologies to try 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.

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 D.22 on page D-45 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 D.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.

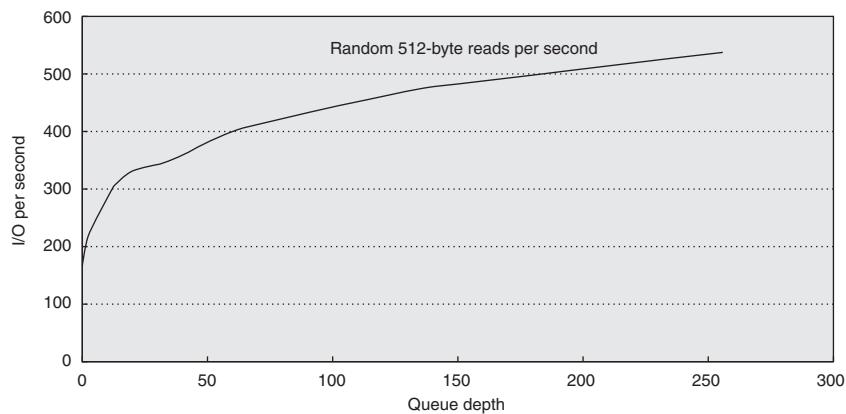


Figure D.2 Throughput versus command queue depth using random 512-byte reads. The disk performs 170 reads per second starting at no command queue and doubles performance at 50 and triples at 256 [Anderson 2003].

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 because 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 2011 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 et al. 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 D.3 shows the specifications of two 3.5-inch disks in 2011. The *Serial ATA* (SATA) disks shoot for high capacity and the best cost per gigabyte, so the 2000 GB drives cost less than \$0.05 per gigabyte. They use the widest platters that fit the form factor and use four or five of them, but they spin at 5900 RPM and seek relatively slowly to allow a higher areal density and to lower power. The corresponding *Serial Attach SCSI* (SAS) drive aims at performance, so it spins at 15,000 RPM and seeks much faster. It uses a lower areal density to spin at that high rate. To reduce power, the platter is much narrower than the form factor. This combination reduces capacity of the SAS drive to 600 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	2000	\$85	4	5900	3.7	16	12	47	45–95	300	32	0.6 M
SAS	600	\$400	4	15,000	2.6	3–4	16	285	122–204	750	16	1.6 M

Figure D.3 Serial ATA (SATA) versus Serial Attach SCSI (SAS) drives in 3.5-inch form factor in 2011. The I/Os per second were 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*, which originally stood 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, measured as either megabytes per second or I/Os per second, make 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 D.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 it 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 (ECCs) 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 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

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 RAID levels; 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	$\sim \log 2$ 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 $\rightarrow 4$ disk accesses	Widely used
6	Row-diagonal parity, EVEN-ODD	2 failures, 2 check disks	Protects against 2 disk failures	Small writes $\rightarrow 6$ disk accesses; $2 \times$ check overhead	Network Appliance

Figure D.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 non-redundant disk array is often called *RAID 0*, indicating that 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 fail. In 2011, 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.

contains the parity of the information in the data disks, a single disk allows recovery from a disk failure. The data are organized in stripes, with N data blocks and one parity block. When a failure occurs, we 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 that the data are 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 are 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 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 (NetApp), 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 et al. 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, so $p - 1$ disks have data. Figure D.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 D.5.

Let's see how row-diagonal parity works by assuming that data disks 1 and 3 fail in Figure D.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, so we next restore the block from diagonal 2 from failed disk 1. When the data for those blocks have been recovered, then the standard RAID recovery scheme can be used to

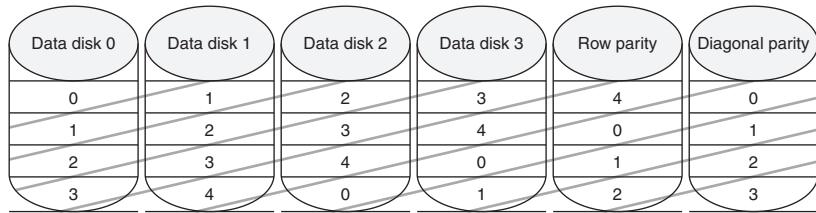


Figure D.5 Row diagonal parity for $p = 5$, which protects four data disks from double failures [Corbett et al. 2004]. This figure shows the diagonal groups for which parity is calculated and stored in the diagonal parity disk. Although this shows all the check data in separate disks for row parity and 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 that 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 up to 256 data disks.

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, Bruck, and Vardy 1996; Blaum et al. 2001].

D.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 relationship among 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; and (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 classified 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 in 2000. This database consisted of high-quality images of over 70,000 artworks [Talagala et al., 2000]. 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 D.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 such as cables would scarcely ever fail.

Figure D.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

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 D.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, hosted 368 8.4 GB, 7200 RPM, 3.5-inch IBM disks. The PCs were P6-200 MHz with 96 MB of DRAM each. They ran FreeBSD 3.0, and the hosts were connected via switched 100 Mbit/sec Ethernet. All SCSI disks were connected to two PCs via double-ended SCSI chains to support RAID 1. The primary application was called the Zoom Project, which in 1998 was the world's largest art image database, with 72,000 images. See Talagala et al. [2000b].

system to boot the PCs.) Figure D.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 D.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 D.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 required yearly service by Tandem, but they were replaced by disks that required 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 1989 was the major source of reported outages (62%), followed by system operations (15%).

The problem with any such statistics is that the 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 are very difficult to collect because operators must report personal mistakes, which may affect the opinion of their managers, which in turn can affect job security and pay raises. Gray suggested that 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

D-14 ■ Appendix D *Storage Systems*

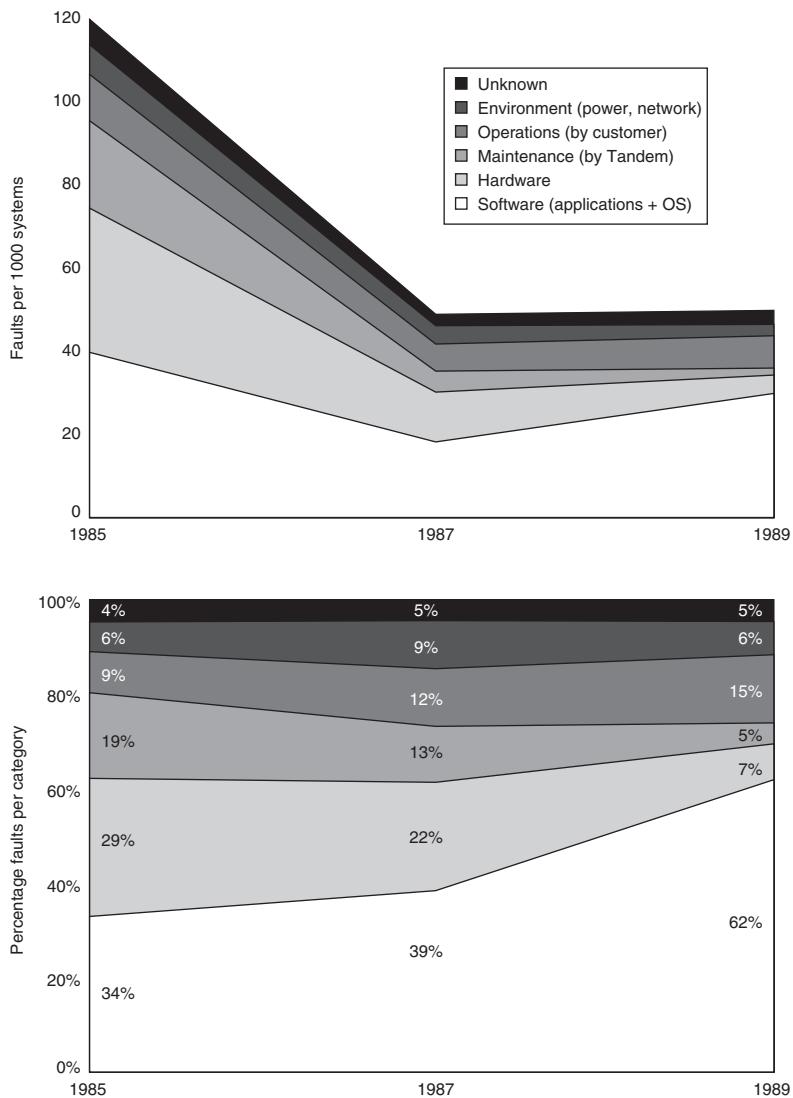


Figure D.7 Faults in Tandem between 1985 and 1989. Gray [1990] collected these data for fault-tolerant Tandem Computers based on reports of component failures by customers.

operator faults by having the system automatically prompt the operator on each 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 believed that operator error is under-reported,

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 2011 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 processes 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.

D.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

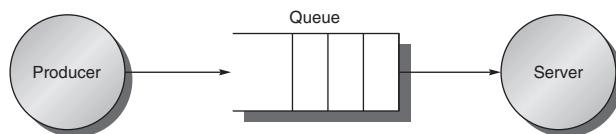


Figure D.8 The traditional producer-server model of response time and throughput. Response time begins when a task is placed in the buffer and ends when it is completed by the server. Throughput is the number of tasks completed by the server in unit time.

against each other. Figure D.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.

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, 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 D.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 D.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 D.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

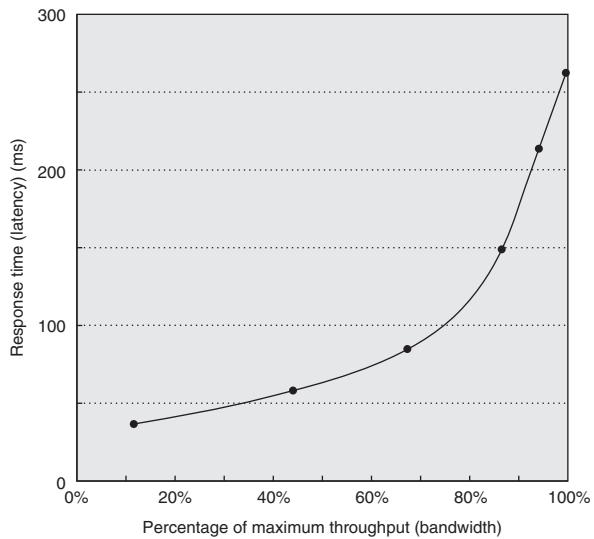


Figure D.9 Throughput versus response time. Latency is normally reported as response time. Note that the minimum response time achieves only 11% of the throughput, while the response time for 100% throughput takes seven times the minimum response time. Note also that the independent variable in this 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.

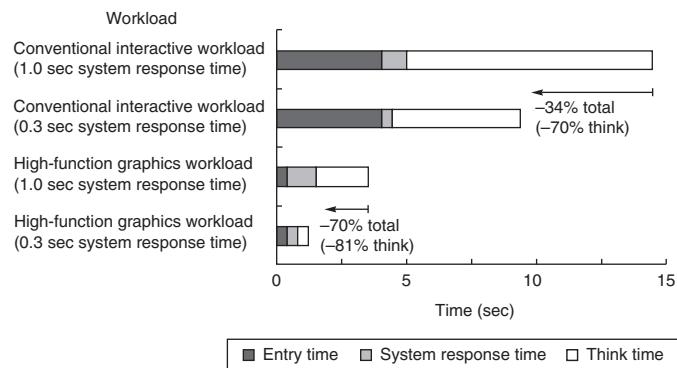


Figure D.10 A user transaction with an interactive computer divided into entry time, system response time, and user think time for a conventional system and graphics system. The entry times are the same, independent of system response time. The entry time was 4 seconds for the conventional system and 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	$\geq 90\%$ of transaction must meet response time limit; 5 seconds for most types of transactions	New order transactions per minute
TPC-W: Transactional Web benchmark	$\geq 90\%$ of Web interactions must meet response time limit; 3 seconds for most types of Web interactions	Web interactions per second
SPECfs97	Average response time ≤ 40 ms	NFS operations per second

Figure D.11 Response time restrictions for three I/O benchmarks.

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 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 D.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, that 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 D.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

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 D.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.

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 transactions per minute (tpmC) and in price of system, including hardware, software, and three years of maintenance support. Figure 1.17 on page 42 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 dataset 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 D.13

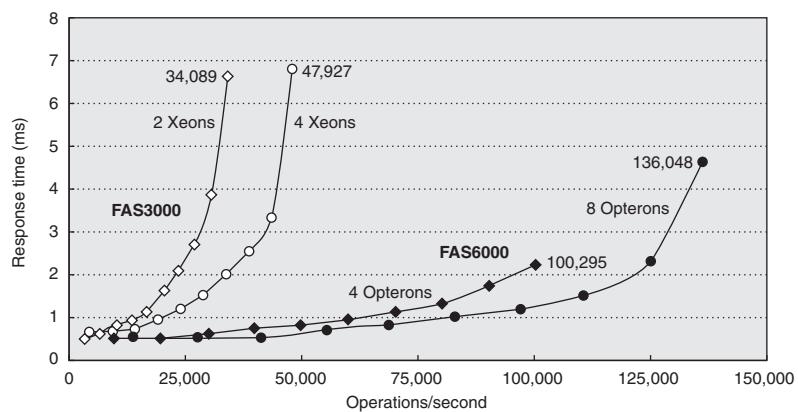


Figure D.13 SPEC SFS97_R1 performance for the NetApp FAS3050c NFS servers in two configurations. Two processors reached 34,089 operations per second and four processors did 47,927. Reported in May 2005, these systems used the Data ONTAP 7.0.1R1 operating system, 2.8 GHz Pentium Xeon microprocessors, 2 GB of DRAM per processor, 1 GB of nonvolatile memory per system, and 168 15 K RPM, 72 GB, Fibre Channel disks. These disks were connected using two or four QLogic ISP-2322 FC disk controllers.

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 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] proposed 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 D.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.

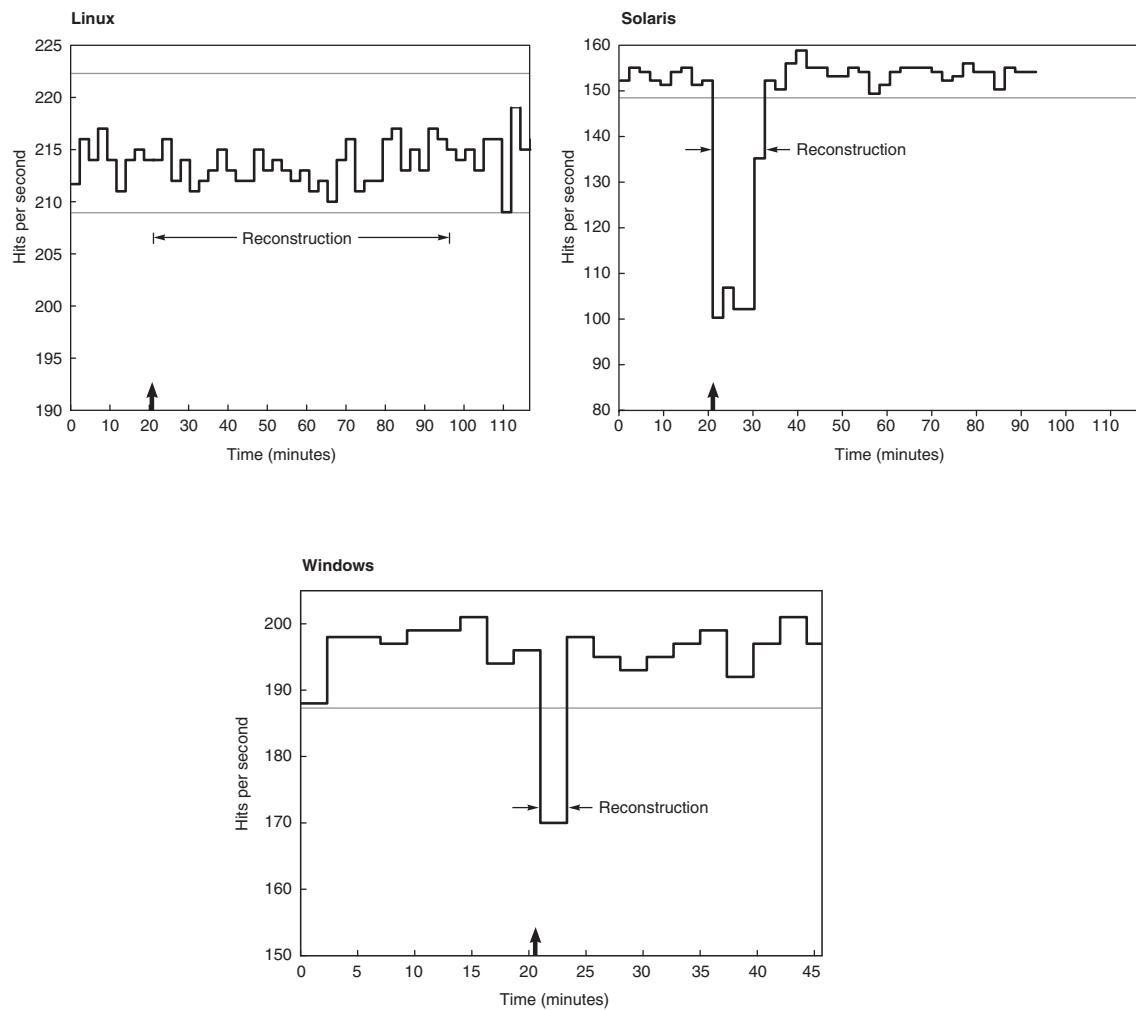


Figure D.14 Availability benchmark for software RAID systems on the same computer running Red Hat 6.0 Linux, Solaris 7, and Windows 2000 operating systems. Note the difference in philosophy on speed of reconstruction of Linux versus Windows and Solaris. The y-axis is behavior in hits per second running 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.

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.

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.

D.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 bestcase 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 shown in Figure D.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 than simulation, we won't cover such analyses here. (See the references in Appendix M for more details.)



Figure D.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.

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:

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

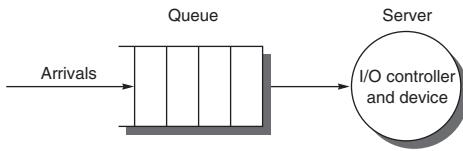


Figure D.16 The single-server model for this section. In this situation, an I/O request “departs” by being completed by the server.

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 D.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.

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 \times $\text{Time}_{\text{server}}$, 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: 50

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

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 D-28.)

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

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

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

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

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 Siméon 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 $a = \text{Rate of events} \times \text{Elapsed time}$. If interarrival times are exponentially distributed and we use the 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 $a = \text{Arrival rate} \times t$. As mentioned on page D-26, 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 \times Average residual service time. This leads to our initial formula:

$$\begin{aligned}\text{Time}_{\text{queue}} &= \text{Length}_{\text{queue}} \times \text{Time}_{\text{server}} \\ &\quad + \text{Server utilization} \times \text{Average residual service time}\end{aligned}$$

Replacing the average residual service time by its definition and $\text{Length}_{\text{queue}}$ by Arrival rate \times $\text{Time}_{\text{queue}}$ 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 $\text{Arrival rate} \times \text{Time}_{\text{server}}$ by $\text{Server utilization}$:

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

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 $\text{Arrival rate} \times \text{Time}_{\text{server}} = \text{Server utilization}$, 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 D-26, 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, 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 2). 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

I = 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:

$$\begin{aligned}\text{Time}_{\text{queue}} &= \text{Time}_{\text{server}} \times \frac{\text{Server utilization}}{(1 - \text{Server utilization})} \\ &= 20 \text{ ms} \times \frac{0.8}{1 - 0.8} = 20 \times \frac{0.8}{0.2} = 20 \times 4 = 80 \text{ ms}\end{aligned}$$

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:

$$\begin{aligned}\text{Time}_{\text{queue}} &= \text{Time}_{\text{server}} \times \frac{\text{Server utilization}}{(1 - \text{Server utilization})} \\ &= 10 \text{ ms} \times \frac{0.4}{1 - 0.4} = 10 \times \frac{0.4}{0.6} = 10 \times \frac{2}{3} = 6.7 \text{ ms}\end{aligned}$$

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.

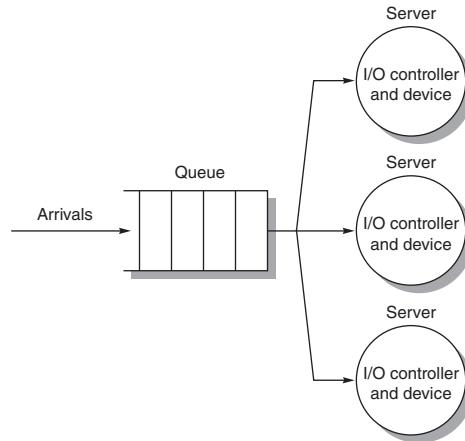


Figure D.17 The M/M/m multiple-server model.

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 D.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:

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

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

The time waiting in the queue is

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

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

$$\text{Prob}_0 \text{ tasks} = \left[1 + \frac{(N_{\text{servers}} \times \text{Utilization})^{N_{\text{servers}}}}{N_{\text{servers}}! \times (1 - \text{Utilization})} + \sum_{n=1}^{N_{\text{servers}}-1} \frac{(N_{\text{servers}} \times \text{Utilization})^n}{n!} \right]^{-1}$$

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

$$\text{Prob}_{\text{tasks} \geq N_{\text{servers}}} = \frac{N_{\text{servers}} \times \text{Utilization}^{N_{\text{servers}}}}{N_{\text{servers}}! \times (1 - \text{Utilization})} \times \text{Prob}_0 \text{ tasks}$$

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.

D.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 2011 has spread itself throughout storage. Figure D.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.

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 D.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.

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 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, we must now ask: 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 Appliance 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

Disk 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.

D.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 sometime in the past. It contains over a petabyte (10^{15} bytes) and is growing by 20 terabytes (10^{12} bytes) of new data per month, so expandible 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 works 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 used 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 D.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



Figure D.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 the back side is used only for airflow. This allows two racks to be placed back-to-back, which reduces the floor space demands in machine rooms.

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.

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 cost \$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 cost \$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 that 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:

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

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

$$\begin{aligned}\text{Maximum IOPS for main memory} &= \frac{266 \times 8}{16 \text{ KB per I/O}} \approx 133,000 \text{ IOPS} \\ \text{Maximum IOPS for main memory} &= \frac{266 \times 8}{50 \text{ KB per I/O}} \approx 42,500 \text{ IOPS}\end{aligned}$$

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

$$\begin{aligned}\text{Maximum IOPS for the I/O bus} &= \frac{133 \text{ MB/sec}}{16 \text{ KB per I/O}} \approx 8300 \text{ IOPS} \\ \text{Maximum IOPS for the I/O bus} &= \frac{133 \text{ MB/sec}}{50 \text{ KB per I/O}} \approx 2700 \text{ IOPS}\end{aligned}$$

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

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

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

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 to 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 $12,320 \times 16 \text{ K} \times 8$ or 1.6 Gbits/sec for 16 KB blocks and $11,680 \times 50 \text{ K} \times 8$ 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 \times (\$500 + 4 \times \$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 to 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 such as 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, for an MTTF of about 125,000 to 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:

$$\begin{aligned} \text{Failure rate} &= \frac{40}{1,000,000} + \frac{160}{125,000} + \frac{40}{500,000} + \frac{1}{500,000} + \frac{40}{200,000} + \frac{40}{200,000} + \frac{80}{1,000,000} \\ &= \frac{40 + 1280 + 80 + 2 + 200 + 200 + 80}{1,000,000 \text{ hours}} = \frac{1882}{1,000,000 \text{ hours}} \end{aligned}$$

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.

D.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 D-8). 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 at a 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 5, 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 4, 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
- Standard I/O options such as 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 $8 \times 6 \times 14$ 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, 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 if running 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.

D.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 D.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 D.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

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 D.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
5000	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 D.21 Minutes unavailable per year to achieve availability class. (From Gray and Siewiorek [1991].) Note that five nines mean unavailable five minutes per year.

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, 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 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 such as 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 D.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 is a large number of tracks. In the past, manufacturers listed the seek of this distance to offer a consistent basis for comparison. (Today, they

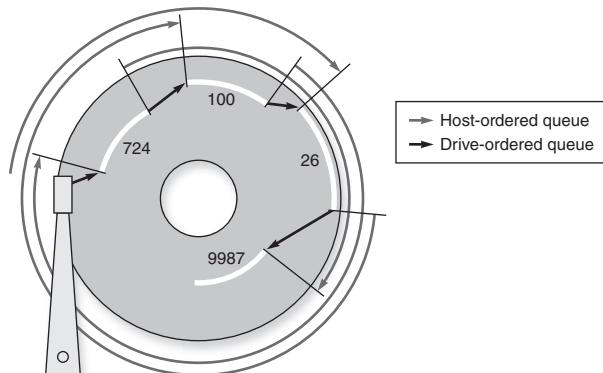


Figure D.22 Example showing OS versus disk schedule accesses, labeled host-ordered versus drive-ordered. The former takes 3 revolutions to complete the 4 reads, while the latter completes them in just 3/4 of a revolution. (From Anderson [2003].)

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

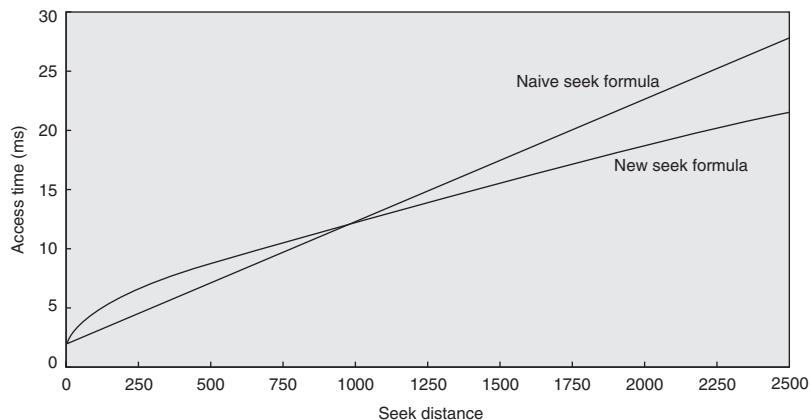
$$\text{Time}_{\text{seek}} = \text{Time}_{\text{minimum}} + \frac{\text{Distance}}{\text{Distance}_{\text{average}}} \times (\text{Time}_{\text{average}} - \text{Time}_{\text{minimum}})$$

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:

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

where a , b , and c are selected for a particular disk so that this formula will match the quoted times for $\text{Distance} = 1$, $\text{Distance} = \text{max}$, and $\text{Distance} = 1/3 \text{ max}$. Figure D.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 temporal and spatial locality (see page B-2 in Appendix B). For example, Figure D.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



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

Figure D.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.

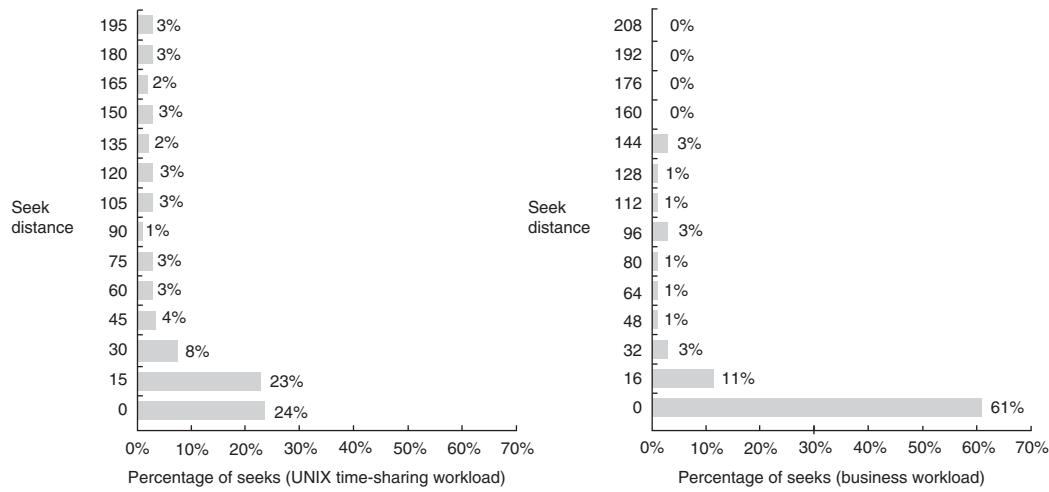


Figure D.24 Sample measurements of seek distances for two systems. The measurements on the left were taken on a UNIX time-sharing system. The measurements on the right were taken from a business-processing application in which the disk seek activity was scheduled to improve throughput. Seek distance 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.

accesses to the same cylinder, labeled distance 0 in the graphs, in both workloads. Thus, this fallacy couldn't be more misleading.

D.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 challenges for storage systems today are 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.

D.11

Historical Perspective and References

Section M.9 (available online) 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 the University of California–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 and Patterson [1999].)

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

- D.1 [10/10/10/10/10] <D.2> The results of running Skippy are shown for a mock disk (Disk Alpha) in Figure D.25.
 - a. [10] <D.2> What is the minimal transfer time?
 - b. [10] <D.2> What is the rotational latency?
 - c. [10] <D.2> What is the head switch time?
 - d. [10] <D.2> What is the cylinder switch time?
 - e. [10] <D.2> What is the number of disk heads?
- D.2 [25] <D.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

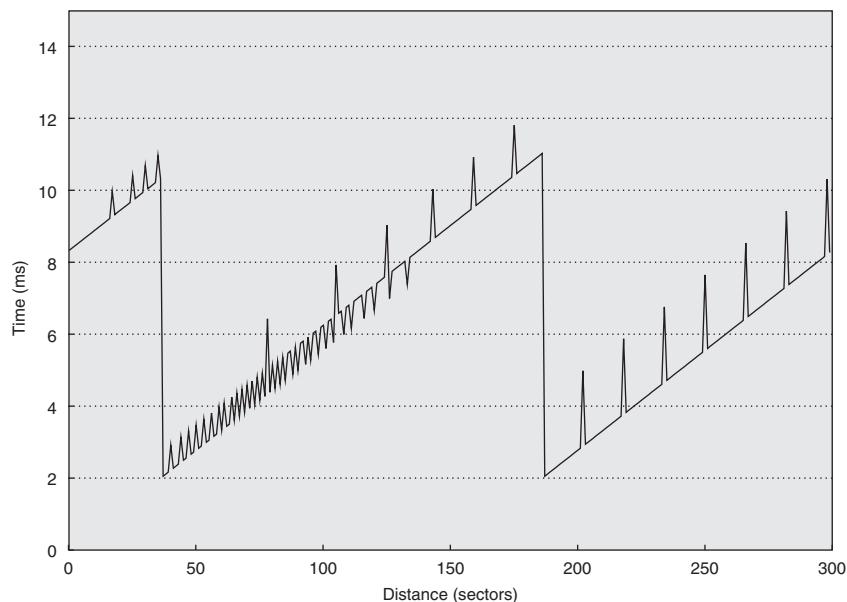


Figure D.25 Results from running Skippy on Disk Alpha.

- Head switch time, 1.0 ms
 - Cylinder switch time, 1.5 ms
 - Number of disk heads, 4
 - Sectors per track, 100
- D.3 [10/10/10/10/10/10] < D.2 > Implement and run the Skippy algorithm on a disk drive of your choosing.
- [10] < D.2 > Graph the results of running Skippy. Report the manufacturer and model of your disk.
 - [10] < D.2 > What is the minimal transfer time?
 - [10] < D.2 > What is the rotational latency?
 - [10] < D.2 > What is the head switch time?
 - [10] < D.2 > What is the cylinder switch time?
 - [10] < D.2 > What is the number of disk heads?
 - [10] < D.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 are 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 .

- D.4 [20/20]< D.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 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]< D.2> Figure D.26 shows the results of running the pattern size algorithm on an unknown RAID system.

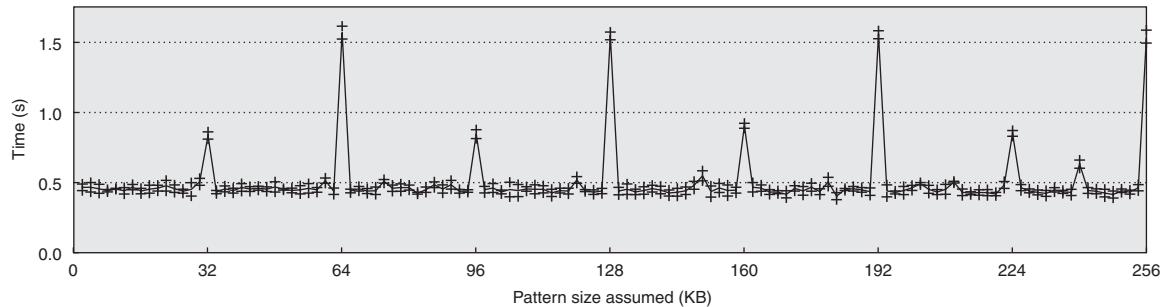


Figure D.26 Results from running the pattern size algorithm of Shear on a mock storage system.

- 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]< D.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
- D.5 [20/20]< D.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;
    }
    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]<D.2>Figure D.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]<D.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

- D.6 [10/10/10/10]<D.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;
```

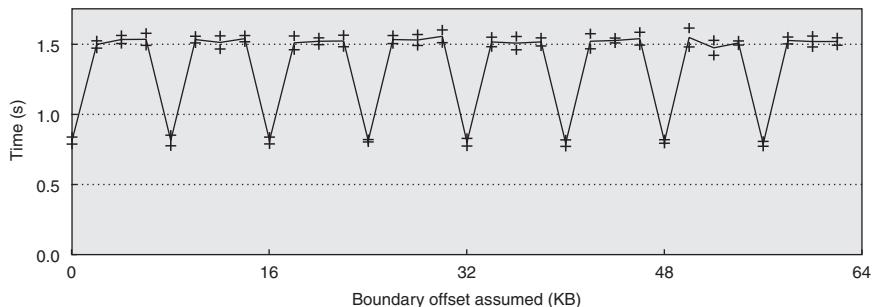


Figure D.27 Results from running the chunk size algorithm of Shear on a mock storage system.

```

        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 D.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]<D.2> How many chunks are in a pattern?
 - b. [20]<D.2> Which chunks of each pattern appear to be allocated on the same disks?
 - c. [20]<D.2> How many disks appear to be in this storage system?
 - d. [20]<D.2> Draw the likely layout of blocks across the disks.
- D.7 [20]<D.2> Draw the graph that would result from running the layout algorithm on the storage system shown in Figure D.29. This storage system has four disks and a chunk size of four 4 KB blocks (16 KB) and is using a RAID 5 Left-Asymmetric layout.

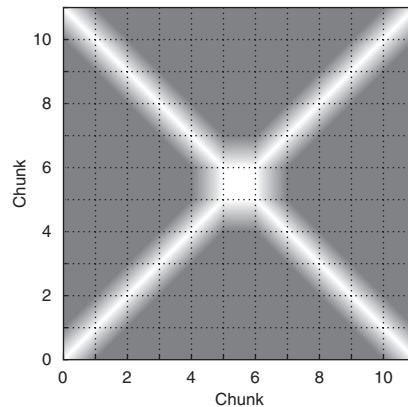
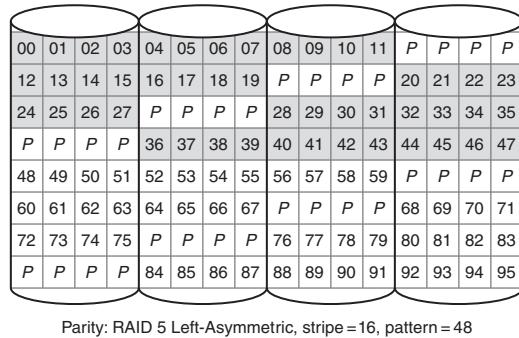


Figure D.28 Results from running the layout algorithm of Shear on a mock storage system.



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

Figure D.29 A storage system with four disks, a chunk size of four 4 KB blocks, and using a RAID 5 Left-Asymmetric layout. Two repetitions of the pattern are shown.

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 are not lost when a disk fails. Thus, one of the key responsibilities of a RAID is to reconstruct the data that were 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 *offline reconstruction*, the RAID devotes all of its resources to performing reconstruction and does not service any requests from the workload. In *online 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 are lost if a second disk fails before the data from the first disk can be 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 that each disk is the 37 GB SCSI disk shown in Figure D.3 and that 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.

- D.8 [10]< D.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.
- D.9 [10]< D.2, D.4 > When a single disk fails, the RAID 4 system will perform reconstruction. What is the expected time until a reconstruction is needed?
- D.10 [10/10/10]< D.2, D.4 > Assume that reconstruction of the RAID 4 array begins at time t .
 - a. [10]< D.2, D.4 > What read and write operations are required to perform the reconstruction?
 - b. [10]< D.2, D.4 > For offline reconstruction, when will the reconstruction process be complete?
 - c. [10]< D.2, D.4 > For online reconstruction, when will the reconstruction process be complete?
- D.11 [10/10/10/10]< D.2, D.4 > In this exercise, we will investigate the mean time until data loss (MTDL). In RAID 4, data are lost only if a second disk fails before the first failed disk is repaired.
 - a. [10]< D.2, D.4 > What is the likelihood of having a second failure during offline reconstruction?
 - b. [10]< D.2, D.4 > Given this likelihood of a second failure during reconstruction, what is the MTDL for offline reconstruction?
 - c. [10]< D.2, D.4 > What is the likelihood of having a second failure during online reconstruction?
 - d. [10]< D.2, D.4 > Given this likelihood of a second failure during reconstruction, what is the MTDL for online reconstruction?
- D.12 [10]< D.2, D.4 > What is performability for the RAID 4 array for offline reconstruction? Calculate the performability using IOPS, assuming a random readonly workload that is evenly distributed across the disks of the RAID 4 array.

- D.13 [10]< D.2, D.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?
- D.14 [10]< D.2, D.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 D.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 $5 \text{ ms} * \text{space utilization}$. 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.

- D.15 [10/10/10/10/10] < D.5 > Begin by assuming that the storage system consists of a single 40 GB disk.
- [10] < D.5 > Given this workload and storage system, what is the average service time?
 - [10] < D.5 > On average, what is the utilization of the disk?
 - [10] < D.5 > On average, how much time does each request spend waiting for the disk?
 - [10] < D.5 > What is the mean number of requests in the queue?
 - [10] < D.5 > Finally, what is the average response time for the disk requests?
- D.16 [10/10/10/10/10] < D.2, D.5 > Imagine that the storage system is now configured to contain two 40 GB disks in a RAID 0 array; that is, the data are striped in blocks of 8 KB equally across the two disks with no redundancy.
- [10] < D.2, D.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?
 - [10] < D.2, D.5 > How can queuing theory be used to model this storage system?
 - [10] < D.2, D.5 > What is the average utilization of each disk?
 - [10] < D.2, D.5 > On average, how much time does each request spend waiting for the disk?
 - [10] < D.2, D.5 > What is the mean number of requests in each queue?
 - [10] < D.2, D.5 > Finally, what is the average response time for the disk requests?
- D.17 [20/20/20/20/20] < D.2, D.5 > Instead imagine that the storage system is configured to contain two 40 GB disks in a RAID 1 array; that is, the data are mirrored

across the two disks. Use queuing theory to model this system for a read-only workload.

- a. [20]<D.2, D.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]<D.2, D.5> How can queuing theory be used to model this storage system?

- c. [20]<D.2, D.5> What is the average utilization of each disk?

- d. [20]<D.2, D.5> On average, how much time does each request spend waiting for the disk?

- e. [20]<D.2, D.5> Finally, what is the average response time for the disk requests?

- D.18 [10/10]<D.2, D.5> Imagine that instead of a read-only workload, you now have a write-only workload on a RAID 1 array.

- a. [10]<D.2, D.5> Describe how you can use queuing theory to model this system and workload.

- b. [10]<D.2, D.5> Given this system and workload, what are 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 D.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.

- D.19 [10]< D.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.
- D.20 [20/20/20/20]< D.2, D.4, D.7 > What configuration of SCSI disks, controllers, and enclosures results in the best performance given your monetary and capacity constraints?
- a. [20]< D.2, D.4, D.7 > How many IOPS do you expect to deliver with your system?
 - b. [20]< D.2, D.4, D.7 > How much does your system cost?
 - c. [20]< D.2, D.4, D.7 > What is the capacity of your system?
 - d. [20]< D.2, D.4, D.7 > What is the MTTF of your system?
- D.21 [10]< D.2, D.4, D.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 also 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?
- D.22 [20/20/20/20/20]< D.2, D.4, D.7 > Optimizing your RAID 10 or RAID 01 array only for reliability (but staying within your capacity and monetary constraints), what is your RAID configuration?
- a. [20]< D.2, D.4, D.7 > What is the overall MTTF of the components in your system?

- b. [20]<D.2, D.4, D.7> What is the MTDL of your system?
 - c. [20]<D.2, D.4, D.7> What is the usable capacity of this system?
 - d. [20]<D.2, D.4, D.7> How much does your system cost?
 - e. [20]<D.2, D.4, D.7> Assuming a write-only workload, how many IOPS can you expect to deliver?
- D.23 [10]<D.2, D.4, D.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 are 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 then 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.

- D.24 [20/20/20]<D.2> Assume that you will use simple parity protection in Exercises D.24 through D.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 studies by 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 that you have a 37 GB disk volume that is roughly half full and follows that same distribution, and answer the following questions:

- a. [20] <D.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] <D.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] <D.2> Given this file distribution, is the block size you are using to compute checksums too big, too little, or just right?
- D.25 [10/10] <D.2, D.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 are there. The problem with this approach is that files that are not accessed frequently may 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] <D.2, D.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 D.3 (4 ms seek, roughly 100 MB/sec transfer), how often should you scan through files to check and repair their integrity?
 - b. [10] <D.2, D.3> At what bit flip rate does it become impossible to maintain data integrity? Again assume the 20 GB volume and the SCSI disk.
- D.26 [10/10/10] <D.2, D.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] <D.2, D.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] <D.2, D.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]<D.2, D.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 check-sums 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 D.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]<D.2, D.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?

- D.27 [40]<D.2, D.3, D.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 for 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 Anon. 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 dataset in memory, you can read the entire dataset 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 $(\text{input file size})/(\text{memory size})$ 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.

- D.28 [20/20/20] < D.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 that 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 are up to you (assume there is no limit on disks per I/O interconnect).

- a. [20]<D.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]<D.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]<D.4> What is the bottleneck in your system?
- D.29 [25/25/25]<D.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 costeffective 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 D.28 to configure the winning machine.

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

- a. [25]<D.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]<D.4> How does the reading, writing, and sorting time break down with this configuration?
 - c. [25]<D.4> What is the bottleneck in your system?
- D.30 [20/20/20]<D.4, D.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]<D.4, D.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]<D.4, D.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]<D.4, D.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?

- D.31 [40]<D.2, D.4, D.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 the data, and then writes the data 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.

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.

E.1	Introduction	E-2
E.2	Signal Processing and Embedded Applications: The Digital Signal Processor	E-5
E.3	Embedded Benchmarks	E-12
E.4	Embedded Multiprocessors	E-14
E.5	Case Study: The Emotion Engine of the Sony PlayStation 2	E-15
E.6	Case Study: Sanyo VPC-SX500 Digital Camera	E-19
E.7	Case Study: Inside a Cell Phone	E-20
E.8	Concluding Remarks	E-25

E

Embedded Systems

**By Thomas M. Conte
North Carolina State University**

Where a calculator on the ENIAC is equipped with 18,000 vacuum tubes and weighs 30 tons, computers in the future may have only 1,000 vacuum tubes and perhaps weigh 1 1/2 tons.

Popular Mechanics
March 1949

E.1 Introduction

Embedded computer systems—computers lodged in other devices where the presence of the computers is not immediately obvious—are the fastest-growing portion of the computer market. These devices range from everyday machines (most microwaves, most washing machines, printers, network switches, and automobiles contain simple to very advanced embedded microprocessors) to handheld digital devices (such as PDAs, cell phones, and music players) to video game consoles and digital set-top boxes. Although in some applications (such as PDAs) the computers are programmable, in many embedded applications the only programming occurs in connection with the initial loading of the application code or a later software upgrade of that application. Thus, the application is carefully tuned for the processor and system. This process sometimes includes limited use of assembly language in key loops, although time-to-market pressures and good software engineering practice restrict such assembly language coding to a fraction of the application.

Compared to desktop and server systems, embedded systems have a much wider range of processing power and cost—from systems containing low-end 8-bit and 16-bit processors that may cost less than a dollar, to those containing full 32-bit microprocessors capable of operating in the 500 MIPS range that cost approximately 10 dollars, to those containing high-end embedded processors that cost hundreds of dollars and can execute several billions of instructions per second. Although the range of computing power in the embedded systems market is very large, price is a key factor in the design of computers for this space. Performance requirements do exist, of course, but the primary goal is often meeting the performance need at a minimum price, rather than achieving higher performance at a higher price.

Embedded systems often process information in very different ways from general-purpose processors. Typically these applications include deadline-driven constraints—so-called *real-time constraints*. In these applications, a particular computation must be completed by a certain time or the system fails (there are other constraints considered real time, discussed in the next subsection).

Embedded systems applications typically involve processing information as *signals*. The lay term “signal” often connotes radio transmission, and that is true for some embedded systems (e.g., cell phones). But a signal may be an image, a motion picture composed of a series of images, a control sensor measurement, and so on. Signal processing requires specific computation that many embedded processors are optimized for. We discuss this in depth below. A wide range of benchmark requirements exist, from the ability to run small, limited code segments to the ability to perform well on applications involving tens to hundreds of thousands of lines of code.

Two other key characteristics exist in many embedded applications: the need to minimize memory and the need to minimize power. In many embedded applications, the memory can be a substantial portion of the system cost, and it is important to optimize memory size in such cases. Sometimes the application is expected to fit

entirely in the memory on the processor chip; other times the application needs to fit in its entirety in a small, off-chip memory. In either case, the importance of memory size translates to an emphasis on code size, since data size is dictated by the application. Some architectures have special instruction set capabilities to reduce code size. Larger memories also mean more power, and optimizing power is often critical in embedded applications. Although the emphasis on low power is frequently driven by the use of batteries, the need to use less expensive packaging (plastic versus ceramic) and the absence of a fan for cooling also limit total power consumption. We examine the issue of power in more detail later in this appendix.

Another important trend in embedded systems is the use of processor cores together with application-specific circuitry—so-called “core plus ASIC” or “system on a chip” (SOC), which may also be viewed as special-purpose multiprocessors (see Section E.4). Often an application’s functional and performance requirements are met by combining a custom hardware solution together with software running on a standardized embedded processor core, which is designed to interface to such special-purpose hardware. In practice, embedded problems are usually solved by one of three approaches:

1. The designer uses a combined hardware/software solution that includes some custom hardware and an embedded processor core that is integrated with the custom hardware, often on the same chip.
2. The designer uses custom software running on an off-the-shelf embedded processor.
3. The designer uses a digital signal processor and custom software for the processor. *Digital signal processors* are processors specially tailored for signal-processing applications. We discuss some of the important differences between digital signal processors and general-purpose embedded processors below.

Figure E.1 summarizes these three classes of computing environments and their important characteristics.

Real-Time Processing

Often, the performance requirement in an embedded application is a real-time requirement. A *real-time performance requirement* is one where a segment of the application has an absolute maximum execution time that is allowed. For example, in a digital set-top box the time to process each video frame is limited, since the processor must accept and process the frame before the next frame arrives (typically called *hard real-time systems*). In some applications, a more sophisticated requirement exists: The average time for a particular task is constrained as well as is the number of instances when some maximum time is exceeded. Such approaches (typically called *soft real-time*) arise when it is possible to occasionally miss the time constraint on an event, as long as not too many are missed. Real-time

Feature	Desktop	Server	Embedded
Price of system	\$1000–\$10,000	\$10,000–\$10,000,000	\$10–\$100,000 (including network routers at the high end)
Price of microprocessor module	\$100–\$1000	\$200–\$2000 (per processor)	\$0.20–\$200 (per processor)
Microprocessors sold per year (estimates for 2000)	150,000,000	4,000,000	300,000,000 (32-bit and 64-bit processors only)
Critical system design issues	Price-performance, graphics performance	Throughput, availability, scalability	Price, power consumption, application-specific performance

Figure E.1 A summary of the three computing classes and their system characteristics. Note the wide range in system price for servers and embedded systems. For servers, this range arises from the need for very large-scale multiprocessor systems for high-end transaction processing and Web server applications. For embedded systems, one significant high-end application is a network router, which could include multiple processors as well as lots of memory and other electronics. The total number of embedded processors sold in 2000 is estimated to exceed 1 billion, if you include 8-bit and 16-bit microprocessors. In fact, the largest-selling microprocessor of all time is an 8-bit microcontroller sold by Intel! It is difficult to separate the low end of the server market from the desktop market, since low-end servers—especially those costing less than \$5000—are essentially no different from desktop PCs. Hence, up to a few million of the PC units may be effectively servers.

performance tends to be highly application dependent. It is usually measured using kernels either from the application or from a standardized benchmark (see Section E.3).

The construction of a hard real-time system involves three key variables. The first is the rate at which a particular task must occur. Coupled to this are the hardware and software required to achieve that real-time rate. Often, structures that are very advantageous on the desktop are the enemy of hard real-time analysis. For example, branch speculation, cache memories, and so on introduce *uncertainty* into code. A particular sequence of code may execute either very efficiently or very inefficiently, depending on whether the hardware branch predictors and caches “do their jobs.” Engineers must analyze code assuming the *worst-case execution time* (WCET). In the case of traditional microprocessor hardware, if one assumes that *all branches are mispredicted* and *all caches miss*, the WCET is overly pessimistic. Thus, the system designer may end up overdesigning a system to achieve a given WCET, when a much less expensive system would have sufficed.

In order to address the challenges of hard real-time systems, and yet still exploit such well-known architectural properties as branch behavior and access locality, it is possible to change how a processor is designed. Consider branch prediction: Although dynamic branch prediction is known to perform far more accurately than static “hint bits” added to branch instructions, the behavior of static hints is much more predictable. Furthermore, although caches perform better than software-managed on-chip memories, the latter produces predictable memory latencies. In some embedded processors, caches can be converted into software-managed on-chip memories via *line locking*. In this approach, a cache line can be locked in the cache so that it cannot be replaced until the line is unlocked

E.2

Signal Processing and Embedded Applications: The Digital Signal Processor

A digital signal processor (DSP) is a special-purpose processor optimized for executing digital signal processing algorithms. Most of these algorithms, from time-domain filtering (e.g., infinite impulse response and finite impulse response filtering), to convolution, to transforms (e.g., fast Fourier transform, discrete cosine transform), to even forward error correction (FEC) encodings, all have as their kernel the same operation: a multiply-accumulate operation. For example, the discrete Fourier transform has the form:

$$X(k) = \sum_{n=0}^{N-1} x(n)W_N^{kn} \text{ where } W_N^{kn} = e^{j\frac{2\pi kn}{N}} = \cos\left(2\pi\frac{kn}{N}\right) + j\sin\left(2\pi\frac{kn}{N}\right)$$

The discrete cosine transform is often a replacement for this because it does not require complex number operations. Either transform has as its core the *sum of a product*. To accelerate this, DSPs typically feature special-purpose hardware to perform *multiply-accumulate* (MAC). A MAC instruction of “MAC A,B,C” has the semantics of “A = A + B * C.” In some situations, the performance of this operation is so critical that a DSP is selected for an application based solely upon its MAC operation throughput.

DSPs often employ *fixed-point* arithmetic. If you think of integers as having a binary point to the right of the least-significant bit, fixed point has a binary point just to the right of the sign bit. Hence, fixed-point data are fractions between -1 and $+1$.

Example Here are three simple 16-bit patterns:

```
0100 0000 0000 0000
0000 1000 0000 0000
0100 1000 0000 1000
```

What values do they represent if they are two’s complement integers? Fixedpoint numbers?

Answer Number representation tells us that the i th digit to the left of the binary point represents 2^{i-1} and the i th digit to the right of the binary point represents 2^{-i} . First assume these three patterns are integers. Then the binary point is to the far right, so they represent 2^{14} , 2^{11} , and $(2^{14} + 2^{11} + 2^3)$, or 16,384, 2048, and 18,440.

Fixed point places the binary point just to the right of the sign bit, so as fixed point these patterns represent 2^{-1} , 2^{-4} , and $(2^{-1} + 2^{-4} + 2^{-12})$. The fractions are $1/2$, $1/16$, and $(2048 + 256 + 1)/4096$ or $2305/4096$, which represents about 0.50000, 0.06250, and 0.56274. Alternatively, for an n -bit two’s complement,

fixed-point number we could just divide the integer presentation by 2^{n-1} to derive the same results:

$$16,384/32,768 = 1/2, \quad 2048/32,768 = 1/16, \text{ and } 18,440/32,768 = 2305/4096.$$

Fixed point can be thought of as a low-cost floating point. It doesn't include an exponent in every word and doesn't have hardware that automatically aligns and normalizes operands. Instead, fixed point relies on the DSP programmer to keep the exponent in a separate variable and ensure that each result is shifted left or right to keep the answer aligned to that variable. Since this exponent variable is often shared by a set of fixed-point variables, this style of arithmetic is also called *blocked floating point*, since a block of variables has a common exponent.

To support such manual calculations, DSPs usually have some registers that are wider to guard against round-off error, just as floating-point units internally have extra guard bits. Figure E.2 surveys four generations of DSPs, listing data sizes and width of the accumulating registers. Note that DSP architects are not bound by the powers of 2 for word sizes. Figure E.3 shows the size of data operands for the TI TMS320C55 DSP.

In addition to MAC operations, DSPs often also have operations to accelerate portions of communications algorithms. An important class of these algorithms revolve around encoding and decoding *forward error correction codes*—codes in which extra information is added to the digital bit stream to guard against errors in transmission. A code of rate m/n has m information bits for $(m + n)$ check bits. So, for example, a $1/2$ rate code would have 1 information bit per every 2 bits. Such codes are often called *trellis codes* because one popular graphical flow diagram of

Generation	Year	Example DSP	Data width	Accumulator width
1	1982	TI TMS32010	16 bits	32 bits
2	1987	Motorola DSP56001	24 bits	56 bits
3	1995	Motorola DSP56301	24 bits	56 bits
4	1998	TI TMS320C6201	16 bits	40 bits

Figure E.2 Four generations of DSPs, their data width, and the width of the registers that reduces round-off error.

Data size	Memory operand in operation	Memory operand in data transfer
16 bits	89.3%	89.0%
32 bits	10.7%	11.0%

Figure E.3 Size of data operands for the TMS320C55 DSP. About 90% of operands are 16 bits. This DSP has two 40-bit accumulators. There are no floating-point operations, as is typical of many DSPs, so these data are all fixed-point integers.

their encoding resembles a garden trellis. A common algorithm for decoding trellis codes is due to Viterbi. This algorithm requires a sequence of compares and selects in order to recover a transmitted bit's true value. Thus DSPs often have compare-select operations to support Viterbi decode for FEC codes.

To explain DSPs better, we will take a detailed look at two DSPs, both produced by Texas Instruments. The TMS320C55 series is a DSP family targeted toward battery-powered embedded applications. In stark contrast to this, the TMS VeloceTI 320C6x series is a line of powerful, eight-issue VLIW processors targeted toward a broader range of applications that may be less power sensitive.

The TI 320C55

At one end of the DSP spectrum is the TI 320C55 architecture. The C55 is optimized for low-power, embedded applications. Its overall architecture is shown in Figure E.4. At the heart of it, the C55 is a seven-staged pipelined CPU. The stages are outlined below:

- *Fetch stage* reads program data from memory into the instruction buffer queue.
- *Decode stage* decodes instructions and dispatches tasks to the other primary functional units.
- *Address stage* computes addresses for data accesses and branch addresses for program discontinuities.
- *Access 1/Access 2 stages* send data read addresses to memory.
- *Read stage* transfers operand data on the B bus, C bus, and D bus.
- *Execute stage* executes operation in the A unit and D unit and performs writes on the E bus and F bus.

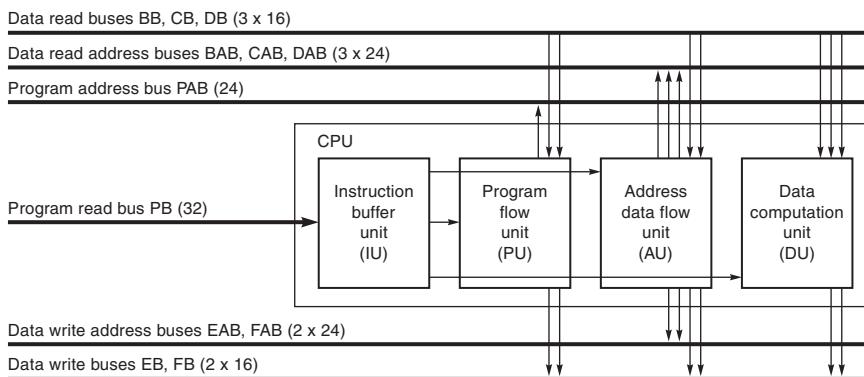


Figure E.4 Architecture of the TMS320C55 DSP. The C55 is a seven-stage pipelined processor with some unique instruction execution facilities. (Courtesy Texas Instruments.)

The C55 pipeline performs pipeline hazard detection and will stall on write after read (WAR) and read after write (RAW) hazards.

The C55 does have a 24 KB instruction cache, but it is configurable to support various workloads. It may be configured to be two-way set associative, direct-mapped, or as a “ramset.” This latter mode is a way to support hard realtime applications. In this mode, blocks in the cache cannot be replaced.

The C55 also has advanced power management. It allows dynamic power management through software-programmable “idle domains.” Blocks of circuitry on the device are organized into these idle domains. Each domain can operate normally or can be placed in a low-power idle state. A programmer-accessible Idle Control Register (ICR) determines which domains will be placed in the idle state when the execution of the next IDLE instruction occurs. The six domains are CPU, direct memory access (DMA), peripherals, clock generator, instruction cache, and external memory interface. When each domain is in the idle state, the functions of that particular domain are not available. However, in the peripheral domain, each peripheral has an Idle Enable bit that controls whether or not the peripheral will respond to the changes in the idle state. Thus, peripherals can be individually configured to idle or remain active when the peripheral domain is idled.

Since the C55 is a DSP, the central feature is its MAC units. The C55 has two MAC units, each comprised of a 17-bit by 17-bit multiplier coupled to a 40-bit dedicated adder. Each MAC unit performs its work in a single cycle; thus, the C55 can execute two MACs per cycle in full pipelined operation. This kind of capability is critical for efficiently performing signal processing applications. The C55 also has a compare, select, and store unit (CSSU) for the add/compare section of the Viterbi decoder.

The TI 320C6x

In stark contrast to the C55 DSP family is the high-end Texas Instruments VeloceTI 320C6x family of processors. The C6x processors are closer to traditional very long instruction word (VLIW) processors because they seek to exploit the high levels of instruction-level parallelism (ILP) in many signal processing algorithms. Texas Instruments is not alone in selecting VLIW for exploiting ILP in the embedded space. Other VLIW DSP vendors include Ceva, StarCore, Philips/TriMedia, and STMicroelectronics. Why do these vendors favor VLIW over superscalar? For the embedded space, code compatibility is less of a problem, and so new applications can be either hand tuned or recompiled for the newest generation of processor. The other reason superscalar excels on the desktop is because the compiler cannot predict memory latencies at compile time. In embedded, however, memory latencies are often much more predictable. In fact, hard real-time constraints force memory latencies to be statically predictable. Of course, a superscalar would also perform well in this environment with these constraints, but the extra hardware to dynamically schedule instructions is both wasteful in terms of precious chip area and in terms of power consumption. Thus VLIW is a natural choice for high-performance embedded.

The C6x family employs different pipeline depths depending on the family member. For the C64x, for example, the pipeline has 11 stages. The first four stages of the pipeline perform instruction fetch, followed by two stages for instruction decode, and finally four stages for instruction execution. The overall architecture of the C64x is shown below in Figure E.5.

The C6x family's execution stage is divided into two parts, the left or "1" side and the right or "2" side. The L1 and L2 units perform logical and arithmetic operations. D units in contrast perform a subset of logical and arithmetic operations but also perform memory accesses (loads and stores). The two M units perform multiplication and related operations (e.g., shifts). Finally the S units perform comparisons, branches, and some SIMD operations (see the next subsection for a detailed explanation of SIMD operations). Each side has its own 32-entry, 32-bit register file (the A file for the 1 side, the B file for the 2 side). A side may access the other side's registers, but with a 1- cycle penalty. Thus, an instruction executing on side 1 may access B5, for example, but it will take 1- cycle extra to execute because of this.

VLIWs are traditionally very bad when it comes to code size, which runs contrary to the needs of embedded systems. However, the C6x family's approach "compresses" instructions, allowing the VLIW code to achieve the same density as equivalent RISC (reduced instruction set computer) code. To do so, instruction fetch is carried out on an "instruction packet," shown in Figure E.6. Each instruction has a p bit that specifies whether this instruction is a member of the current

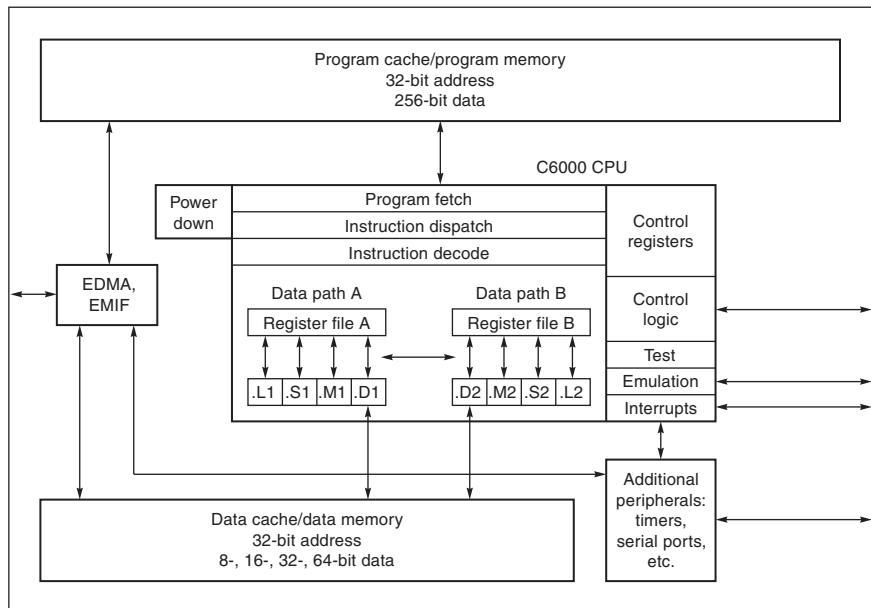


Figure E.5 Architecture of the TMS320C64x family of DSPs. The C6x is an eight-issue traditional VLIW processor. (Courtesy Texas Instruments.)

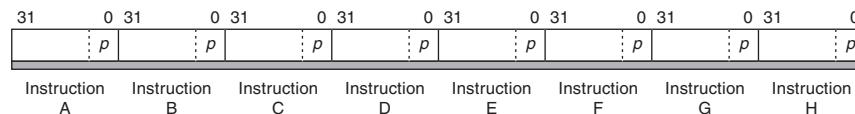


Figure E.6 Instruction packet of the TMS320C6x family of DSPs. The *p* bits determine whether an instruction begins a new VLIW word or not. If the *p* bit of instruction *i* is 1, then instruction *i* + 1 is to be executed in parallel with (in the same cycle as) instruction *i*. If the *p* bit of instruction *i* is 0, then instruction *i* + 1 is executed in the cycle after instruction *i*. (Courtesy Texas Instruments.)

VLIW word or the next VLIW word (see the figure for a detailed explanation). Thus, there are now no NOPs that are needed for VLIW encoding.

Software pipelining is an important technique for achieving high performance in a VLIW. But software pipelining relies on each iteration of the loop having an identical schedule to all other iterations. Because conditional branch instructions disrupt this pattern, the C6x family provides a means to conditionally execute instructions using *predication*. In predication, the instruction performs its work. But when it is done executing, an additional register, for example A1, is checked. If A1 is zero, the instruction does not write its results. If A1 is nonzero, the instruction proceeds normally. This allows simple if-then and if-then-else structures to be collapsed into straight-line code for software pipelining.

Media Extensions

There is a middle ground between DSPs and microcontrollers: *media extensions*. These extensions add DSP-like capabilities to microcontroller architectures at relatively low cost. Because media processing is judged by human perception, the data for multimedia operations are often much narrower than the 64-bit data word of modern desktop and server processors. For example, floating-point operations for graphics are normally in single precision, not double precision, and often at a precision less than is required by IEEE 754. Rather than waste the 64-bit arithmetic-logical units (ALUs) when operating on 32-bit, 16-bit, or even 8-bit integers, multimedia instructions can operate on several narrower data items at the same time. Thus, a *partitioned add* operation on 16-bit data with a 64-bit ALU would perform four 16-bit adds in a single clock cycle. The extra hardware cost is simply to prevent carries between the four 16-bit partitions of the ALU. For example, such instructions might be used for graphical operations on pixels. These operations are commonly called *single-instruction multiple-data* (SIMD) or *vector* instructions.

Most graphics multimedia applications use 32-bit floating-point operations. Some computers double peak performance of single-precision, floating-point operations; they allow a single instruction to launch two 32-bit operations on operands found side by side in a double-precision register. The two partitions must be insulated to prevent operations on one half from affecting the other. Such floating-point operations are called *paired single operations*. For example, such an operation

might be used for graphical transformations of vertices. This doubling in performance is typically accomplished by doubling the number of floating-point units, making it more expensive than just suppressing carries in integer adders.

Figure E.7 summarizes the SIMD multimedia instructions found in several recent computers.

DSPs also provide operations found in the first three rows of Figure E.7, but they change the semantics a bit. First, because they are often used in real-time applications, there is not an option of causing an exception on arithmetic overflow (otherwise it could miss an event); thus, the result will be used no matter what the inputs. To support such an unyielding environment, DSP architectures use *saturating arithmetic*: If the result is too large to be represented, it is set to the largest representable number, depending on the sign of the result. In contrast, two's complement arithmetic can add a small positive number to a large positive.

Instruction category	Alpha MAX	HP PA-RISC MAX2	Intel Pentium MMX	PowerPC AltiVec	SPARC VIS
Add/subtract		4H	8B, 4H, 2W	16B, 8H, 4W	4H, 2W
Saturating add/subtract		4H	8B, 4H	16B, 8H, 4W	
Multiply			4H	16B, 8H	
Compare	8B (\geq)		8B, 4H, 2W (=, >)	16B, 8H, 4W (=, >, \geq , <, \leq)	4H, 2W (=, not =, >, \leq)
Shift right/left		4H	4H, 2W	16B, 8H, 4W	
Shift right arithmetic		4H		16B, 8H, 4W	
Multiply and add				8H	
Shift and add (saturating)		4H			
AND/OR/XOR	8B, 4H, 2W	8B, 4H, 2W	8B, 4H, 2W	16B, 8H, 4W	8B, 4H, 2W
Absolute difference	8B			16B, 8H, 4W	8B
Maximum/minimum	8B, 4W			16B, 8H, 4W	
Pack (2n bits \rightarrow n bits)	2B \rightarrow 2B, 4B \rightarrow 4B	2*4H \rightarrow 8B	4H \rightarrow 4B, 2W \rightarrow 2H	4W \rightarrow 4B, 8H \rightarrow 8B	2W \rightarrow 2H, 2W \rightarrow 2B, 4H \rightarrow 4B
Unpack/merge	2B \rightarrow 2W, 4B \rightarrow 4H		2B \rightarrow 2W, 4B \rightarrow 4H	4B \rightarrow 4W, 8B \rightarrow 8H	4B \rightarrow 4H, 2*4B \rightarrow 8B
Permute/shuffle		4H		16B, 8H, 4W	

Figure E.7 Summary of multimedia support for desktop processors. Note the diversity of support, with little in common across the five architectures. All are fixed-width operations, performing multiple narrow operations on either a 64-bit or 128-bit ALU. B stands for byte (8 bits), H for half word (16 bits), and W for word (32 bits). Thus, 8B means an operation on 8 bytes in a single instruction. Note that AltiVec assumes a 128-bit ALU, and the rest assume 64 bits. Pack and unpack use the notation 2*2W to mean 2 operands each with 2 words. This table is a simplification of the full multimedia architectures, leaving out many details. For example, HP MAX2 includes an instruction to calculate averages, and SPARC VIS includes instructions to set registers to constants. Also, this table does not include the memory alignment operation of AltiVec, MAX, and VIS.

E.3 Embedded Benchmarks

It used to be the case just a couple of years ago that in the embedded market, many manufacturers quoted Dhrystone performance, a benchmark that was criticized and given up by desktop systems more than 20 years ago! As mentioned earlier, the enormous variety in embedded applications, as well as differences in performance requirements (hard real time, soft real time, and overall cost-performance), make the use of a single set of benchmarks unrealistic. In practice, many designers of embedded systems devise benchmarks that reflect their application, either as kernels or as stand-alone versions of the entire application.

For those embedded applications that can be characterized well by kernel performance, the best standardized set of benchmarks appears to be a new benchmark set: the EDN Embedded Microprocessor Benchmark Consortium (or EEMBC, pronounced “embassy”). The EEMBC benchmarks fall into six classes (called “subcommittees” in the parlance of EEMBC): automotive/industrial, consumer, telecommunications, digital entertainment, networking (currently in its second version), and office automation (also the second version of this subcommittee). Figure E.8 shows the six different application classes, which include 50 benchmarks.

Although many embedded applications are sensitive to the performance of small kernels, remember that often the overall performance of the entire application (which may be thousands of lines) is also critical. Thus, for many embedded systems, the EMBCC benchmarks can only be used to partially assess performance.

Benchmark type (“subcommittee”)	Number of kernels	Example benchmarks
Automotive/industrial	16	6 microbenchmarks (arithmetic operations, pointer chasing, memory performance, matrix arithmetic, table lookup, bit manipulation), 5 automobile control benchmarks, and 5 filter or FFT benchmarks
Consumer	5	5 multimedia benchmarks (JPEG compress/decompress, filtering, and RGB conversions)
Telecommunications	5	Filtering and DSP benchmarks (autocorrelation, FFT, decoder, encoder)
Digital entertainment	12	MP3 decode, MPEG-2 and MPEG-4 encode and decode (each of which is applied to five different datasets), MPEG Encode Floating Point, 4 benchmark tests for common cryptographic standards and algorithms (AES, DES, RSA, and Huffman decoding for data decompression), and enhanced JPEG and color-space conversion tests
Networking version 2	6	IP Packet Check (borrowed from the RFC1812 standard), IP Reassembly, IP Network Address Translator (NAT), Route Lookup, OSPF, Quality of Service (QOS), and TCP
Office automation version 2	6	Ghostscript, text parsing, image rotation, dithering, Bézier

Figure E.8 The EEMBC benchmark suite, consisting of 50 kernels in six different classes. See www.eembc.org for more information on the benchmarks and for scores.

Power Consumption and Efficiency as the Metric

Cost and power are often at least as important as performance in the embedded market. In addition to the cost of the processor module (which includes any required interface chips), memory is often the next most costly part of an embedded system. Unlike a desktop or server system, most embedded systems do not have secondary storage; instead, the entire application must reside in either FLASH or DRAM. Because many embedded systems, such as PDAs and cell phones, are constrained by both cost and physical size, the amount of memory needed for the application is critical. Likewise, power is often a determining factor in choosing a processor, especially for battery-powered systems.

EEMBC EnergyBench provides data on the amount of energy a processor consumes while running EEMBC's performance benchmarks. An EEMBC-certified Energymark score is an optional metric that a device manufacturer may choose to supply in conjunction with certified scores for device performance as a way of indicating a processor's efficient use of power and energy. EEMBC has standardized on the use of National Instruments' LabVIEW graphical development environment and data acquisition hardware to implement EnergyBench.

Figure E.9 shows the relative performance per watt of typical operating power. Compare this figure to Figure E.10, which plots raw performance, and notice how different the results are. The NEC VR 4122 has a clear advantage in performance per watt, but is the second-lowest performing processor! From the viewpoint of power consumption, the NEC VR 4122, which was designed for battery-based systems, is the big winner. The IBM PowerPC displays efficient use of power to achieve its high performance, although at 6 W typical, it is probably not suitable for most battery-based devices.

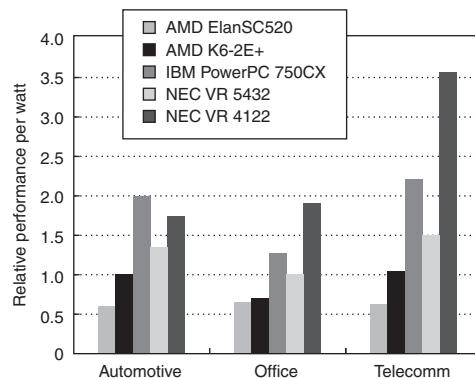


Figure E.9 Relative performance per watt for the five embedded processors. The power is measured as typical operating power for the processor and does not include any interface chips.

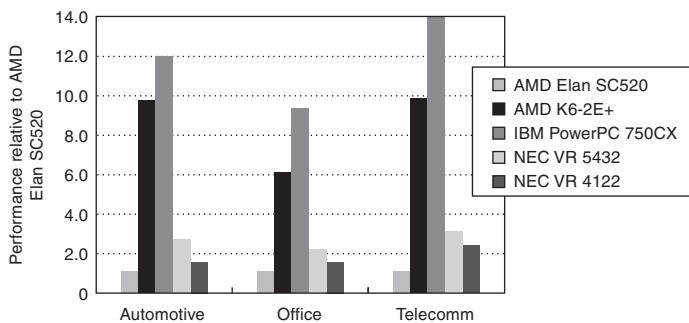


Figure E.10 Raw performance for the five embedded processors. The performance is presented as relative to the performance of the AMD ElanSC520.

E.4

Embedded Multiprocessors

Multiprocessors are now common in server environments, and several desktop multiprocessors are available from vendors, such as Sun, Compaq, and Apple. In the embedded space, a number of special-purpose designs have used customized multiprocessors, including the Sony PlayStation 2 (see Section E.5).

Many special-purpose embedded designs consist of a general-purpose programmable processor or DSP with special-purpose, finite-state machines that are used for stream-oriented I/O. In applications ranging from computer graphics and media processing to telecommunications, this style of special-purpose multiprocessor is becoming common. Although the interprocessor interactions in such designs are highly regimented and relatively simple—consisting primarily of a simple communication channel—because much of the design is committed to silicon, ensuring that the communication protocols among the input/output processors and the general-purpose processor are correct is a major challenge in such designs.

More recently, we have seen the first appearance, in the embedded space, of embedded multiprocessors built from several general-purpose processors. These multiprocessors have been focused primarily on the high-end telecommunications and networking market, where scalability is critical. An example of such a design is the MXP processor designed by empowerTel Networks for use in voice-over-IP systems. The MXP processor consists of four main components:

- An interface to serial voice streams, including support for handling jitter
- Support for fast packet routing and channel lookup
- A complete Ethernet interface, including the MAC layer
- Four MIPS32 R4000-class processors, each with its own cache (a total of 48 KB or 12 KB per processor)

The MIPS processors are used to run the code responsible for maintaining the voice-over-IP channels, including the assurance of quality of service, echo cancellation, simple compression, and packet encoding. Since the goal is to run as many independent voice streams as possible, a multiprocessor is an ideal solution.

Because of the small size of the MIPS cores, the entire chip takes only 13.5 M transistors. Future generations of the chip are expected to handle more voice channels, as well as do more sophisticated echo cancellation, voice activity detection, and more sophisticated compression.

Multiprocessing is becoming widespread in the embedded computing arena for two primary reasons. First, the issues of binary software compatibility, which plague desktop and server systems, are less relevant in the embedded space. Often software in an embedded application is written from scratch for an application or significantly modified (note that this is also the reason VLIW is favored over superscalar in embedded instruction-level parallelism). Second, the applications often have natural parallelism, especially at the high end of the embedded space. Examples of this natural parallelism abound in applications such as a settop box, a network switch, a cell phone (see Section E.7) or a game system (see Section E.5). The lower barriers to use of thread-level parallelism together with the greater sensitivity to die cost (and hence efficient use of silicon) are leading to widespread adoption of multiprocessing in the embedded space, as the application needs grow to demand more performance.

E.5

Case Study: The Emotion Engine of the Sony PlayStation 2

Desktop computers and servers rely on the memory hierarchy to reduce average access time to relatively static data, but there are embedded applications where data are often a continuous stream. In such applications there is still spatial locality, but temporal locality is much more limited.

To give another look at memory performance beyond the desktop, this section examines the microprocessor at the heart of the Sony PlayStation 2. As we will see, the steady stream of graphics and audio demanded by electronic games leads to a different approach to memory design. The style is high bandwidth via many dedicated independent memories.

Figure E.11 shows a block diagram of the Sony PlayStation 2 (PS2). Not surprisingly for a game machine, there are interfaces for video, sound, and a DVD player. Surprisingly, there are two standard computer I/O buses, USB and IEEE 1394, a PCMCIA slot as found in portable PCs, and a modem. These additions show that Sony had greater plans for the PS2 beyond traditional games. Although it appears that the I/O processor (IOP) simply handles the I/O devices and the game console, it includes a 34 MHz MIPS processor that also acts as the emulation computer to run games for earlier Sony PlayStations. It also connects to a standard PC audio card to provide the sound for the games.

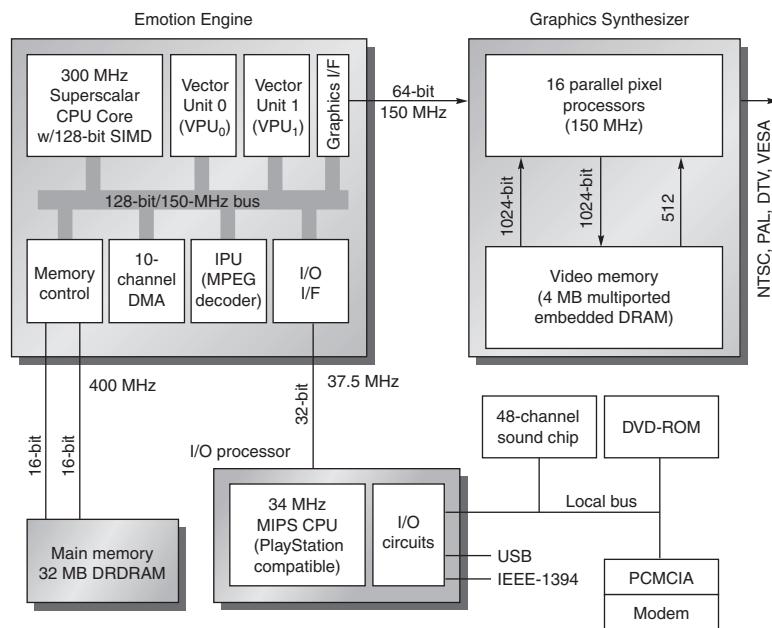


Figure E.11 Block diagram of the Sony PlayStation 2. The 10 DMA channels orchestrate the transfers between all the small memories on the chip, which when completed all head toward the Graphics Interface so as to be rendered by the Graphics Synthesizer. The Graphics Synthesizer uses DRAM on chip to provide an entire frame buffer plus graphics processors to perform the rendering desired based on the display commands given from the Emotion Engine. The embedded DRAM allows 1024-bit transfers between the pixel processors and the display buffer. The Superscalar CPU is a 64-bit MIPS III with two-instruction issue, and comes with a two-way, set associative, 16 KB instruction cache; a two-way, set associative, 8 KB data cache; and 16 KB of scratchpad memory. It has been extended with 128-bit SIMD instructions for multimedia applications (see Section E.2). Vector Unit 0 is primarily a DSP-like coprocessor for the CPU (see Section E.2), which can operate on 128-bit registers in SIMD manner between 8 bits and 32 bits per word. It has 4 KB of instruction memory and 4 KB of data memory. Vector Unit 1 has similar functions to VPU0, but it normally operates independently of the CPU and contains 16 KB of instruction memory and 16 KB of data memory. All three units can communicate over the 128-bit system bus, but there is also a 128-bit dedicated path between the CPU and VPU0 and a 128-bit dedicated path between VPU1 and the Graphics Interface. Although VPU0 and VPU1 have identical microarchitectures, the differences in memory size and units to which they have direct connections affect the roles that they take in a game. At 0.25-micron line widths, the Emotion Engine chip uses 13.5M transistors and is 225 mm², and the Graphics Synthesizer is 279 mm². To put this in perspective, the Alpha 21264 microprocessor in 0.25-micron technology is about 160 mm² and uses 15M transistors. (This figure is based on Figure 1 in "Sony's Emotionally Charged Chip," *Microprocessor Report* 13:5.)

Thus, one challenge for the memory system of this embedded application is to act as source or destination for the extensive number of I/O devices. The PS2 designers met this challenge with two PC800 (400 MHz) DRDRAM chips using two channels, offering 32 MB of storage and a peak memory bandwidth of 3.2 GB/sec.

What's left in the figure are basically two big chips: the Graphics Synthesizer and the Emotion Engine.

The Graphics Synthesizer takes rendering commands from the Emotion Engine in what are commonly called *display lists*. These are lists of 32-bit commands that tell the renderer what shape to use and where to place them, plus what colors and textures to fill them.

This chip also has the highest bandwidth portion of the memory system. By using embedded DRAM on the Graphics Synthesizer, the chip contains the full video buffer *and* has a 2048-bit-wide interface so that pixel filling is not a bottleneck. This embedded DRAM greatly reduces the bandwidth demands on the DRDRAM. It illustrates a common technique found in embedded applications: separate memories dedicated to individual functions to inexpensively achieve greater memory bandwidth for the entire system.

The remaining large chip is the Emotion Engine, and its job is to accept inputs from the IOP and create the display lists of a video game to enable 3D video transformations in real time. A major insight shaped the design of the Emotion Engine: Generally, in a racing car game there are foreground objects that are constantly changing and background objects that change less in reaction to the events, although the background can be most of the screen. This observation led to a split of responsibilities.

The CPU works with VPU0 as a tightly coupled coprocessor, in that every VPU0 instruction is a standard MIPS coprocessor instruction, and the addresses are generated by the MIPS CPU. VPU0 is called a vector processor, but it is similar to 128-bit SIMD extensions for multimedia found in several desktop processors (see Section E.2).

VPU1, in contrast, fetches its own instructions and data and acts in parallel with CPU/VPU0, acting more like a traditional vector unit. With this split, the more flexible CPU/VPU0 handles the foreground action and the VPU1 handles the background. Both deposit their resulting display lists into the Graphics Interface to send the lists to the Graphics Synthesizer.

Thus, the programmers of the Emotion Engine have three processor sets to choose from to implement their programs: the traditional 64-bit MIPS architecture including a floating-point unit, the MIPS architecture extended with multimedia instructions (VPU0), and an independent vector processor (VPU1). To accelerate MPEG decoding, there is another coprocessor (Image Processing Unit) that can act independent of the other two.

With this split of function, the question then is how to connect the units together, how to make the data flow between units, and how to provide the memory bandwidth needed by all these units. As mentioned earlier, the Emotion Engine designers chose many dedicated memories. The CPU has a 16 KB scratch pad memory (SPRAM) in addition to a 16 KB instruction cache and an 8 KB data cache. VPU0 has a 4 KB instruction memory and a 4 KB data memory, and VPU1 has a 16 KB instruction memory and a 16 KB data memory. Note that these are four *memories*, not caches of a larger memory elsewhere. In each memory the latency is just 1 clock cycle. VPU1 has more memory than VPU0 because it creates the bulk of the display lists and because it largely acts independently.

The programmer organizes all memories as two double buffers, one pair for the incoming DMA data and one pair for the outgoing DMA data. The programmer then uses the various processors to transform the data from the input buffer to the output buffer. To keep the data flowing among the units, the programmer next sets up the 10 DMA channels, taking care to meet the real-time deadline for realistic animation of 15 frames per second.

Figure E.12 shows that this organization supports two main operating modes: serial, where CPU/VPU0 acts as a preprocessor on what to give VPU1 for it to create for the Graphics Interface using the scratchpad memory as the buffer, and parallel, where both the CPU/VPU0 and VPU1 create display lists. The display lists and the Graphics Synthesizer have multiple context identifiers to distinguish the parallel display lists to produce a coherent final image.

All units in the Emotion Engine are linked by a common 150 MHz, 128-bit-wide bus. To offer greater bandwidth, there are also two dedicated buses: a 128-bit path between the CPU and VPU0 and a 128-bit path between VPU1 and the Graphics Interface. The programmer also chooses which bus to use when setting up the DMA channels.

Looking at the big picture, if a server-oriented designer had been given the problem, we might see a single common bus with many local caches and cache-coherent mechanisms to keep data consistent. In contrast, the PlayStation 2 followed the tradition of embedded designers and has at least nine distinct memory modules. To keep the data flowing in real time from memory to the display, the PS2 uses dedicated memories, dedicated buses, and DMA channels. Coherency is the responsibility of the programmer, and, given the continuous flow from main memory to the graphics interface and the real-time requirements, programmer-controlled coherency works well for this application.

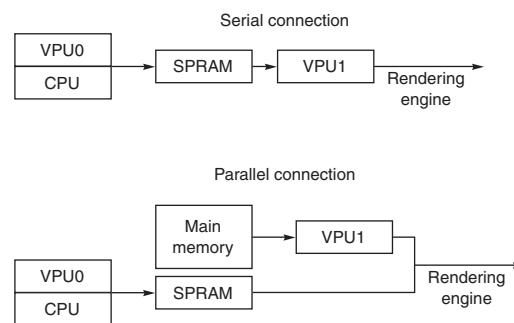


Figure E.12 Two modes of using Emotion Engine organization. The first mode divides the work between the two units and then allows the Graphics Interface to properly merge the display lists. The second mode uses CPU/VPU0 as a filter of what to send to VPU1, which then does all the display lists. It is up to the programmer to choose between serial and parallel data flow. SPRAM is the scratchpad memory.

E.6

Case Study: Sanyo VPC-SX500 Digital Camera

Another very familiar embedded system is a digital camera. Here we consider the Sanyo VPC-SX500. When powered on, the microprocessor of the camera first runs diagnostics on all components and writes any error messages to the liquid crystal display (LCD) on the back of the camera. This camera uses a 1.8-inch low-temperature polysilicon thin-film transistor (TFT) color LCD. When a photographer takes a picture, he first holds the shutter halfway so that the microprocessor can take a light reading. The microprocessor then keeps the shutter open to get the necessary light, which is captured by a charge-coupled device (CCD) as red, green, and blue pixels. The CCD is a 1/2-inch, 1360×1024 -pixel, progressive-scan chip. The pixels are scanned out row by row; passed through routines for white balance, color, and aliasing correction; and then stored in a 4 MB frame buffer. The next step is to compress the image into a standard format, such as JPEG, and store it in the removable Flash memory. The photographer picks the compression, in this camera called either *fine* or *normal*, with a compression ratio of 10 to 20 times. A 512 MB Flash memory can store at least 1200 fine-quality compressed images or approximately 2000 normal-quality compressed images. The microprocessor then updates the LCD display to show that there is room for one less picture.

Although the previous paragraph covers the basics of a digital camera, there are many more features that are included: showing the recorded images on the color LCD display, sleep mode to save battery life, monitoring battery energy, buffering to allow recording a rapid sequence of uncompressed images, and, in this camera, video recording using MPEG format and audio recording using WAV format.

The electronic brain of this camera is an embedded computer with several special functions embedded on the chip [Okada et al. 1999]. Figure E.13 shows the block diagram of a chip similar to the one in the camera. As mentioned in Section E.1, such chips have been called *systems on a chip* (SOCs) because they essentially integrate into a single chip all the parts that were found on a small printed circuit board of the past. A SOC generally reduces size and lowers power compared to less integrated solutions. Sanyo claims their SOC enables the camera to operate on half the number of batteries and to offer a smaller form factor than competitors' cameras. For higher performance, it has two buses. The 16-bit bus is for the many slower I/O devices: SmartMedia interface, program and data memory, and DMA. The 32-bit bus is for the SDRAM, the signal processor (which is connected to the CCD), the Motion JPEG encoder, and the NTSC/PAL encoder (which is connected to the LCD). Unlike desktop microprocessors, note the large variety of I/O buses that this chip must integrate. The 32-bit RISC MPU is a proprietary design and runs at 28.8 MHz, the same clock rate as the buses. This 700 mW chip contains 1.8M transistors in a 10.5×10.5 mm die implemented using a 0.35-micron process.

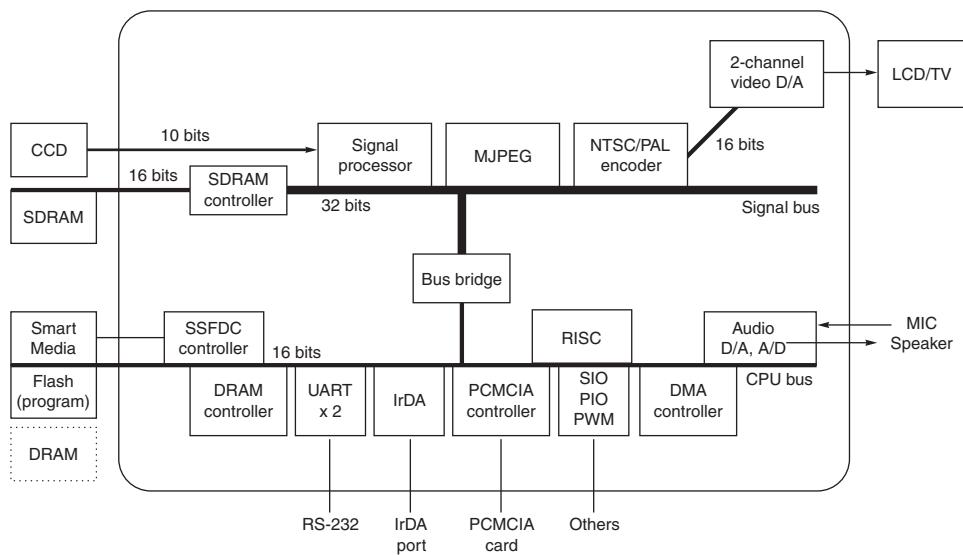


Figure E.13 The system on a chip (SOC) found in Sanyo digital cameras. This block diagram, found in Okada et al. [1999], is for the predecessor of the SOC in the camera described in the text. The successor SOC, called *Super Advanced IC*, uses three buses instead of two, operates at 60 MHz, consumes 800 mW, and fits 3.1M transistors in a 10.2×10.2 mm die using a 0.35-micron process. Note that this embedded system has twice as many transistors as the state-of-the-art, high-performance microprocessor in 1990! The SOC in the figure is limited to processing 1024×768 pixels, but its successor supports 1360×1024 pixels.

E.7

Case Study: Inside a Cell Phone

Although gaming consoles and digital cameras are familiar embedded systems, today the most familiar embedded system is the cell phone. In 1999, there were 76 million cellular subscribers in the United States, a 25% growth rate from the year before. That growth rate is almost 35% per year worldwide, as developing countries find it much cheaper to install cellular towers than copper-wire-based infrastructure. Thus, in many countries, the number of cell phones in use exceeds the number of wired phones in use.

Not surprisingly, the cellular handset market is growing at 35% per year, with about 280 million cellular phone handsets sold worldwide in 1999. To put that in perspective, in the same year sales of personal computers were 120 million. These numbers mean that tremendous engineering resources are available to improve cell phones, and cell phones are probably leaders in engineering innovation per cubic inch [Grice and Kanellos 2000].

Before unveiling the anatomy of a cell phone, let's try a short introduction to wireless technology.

Background on Wireless Networks

Networks can be created out of thin air as well as out of copper and glass, creating *wireless networks*. Much of this section is based on a report from the National Research Council [1997].

A radio wave is an electromagnetic wave propagated by an antenna. Radio waves are modulated, which means that the sound signal is superimposed on the stronger radio wave that carries the sound signal, and hence is called the *carrier signal*. Radio waves have a particular wavelength or frequency: They are measured either as the length of the complete wave or as the number of waves per second. Long waves have low frequencies, and short waves have high frequencies. FM radio stations transmit on the band of 88 MHz to 108 MHz using frequency modulations (FM) to record the sound signal.

By tuning in to different frequencies, a radio receiver can pick up a specific signal. In addition to AM and FM radio, other frequencies are reserved for citizens band radio, television, pagers, air traffic control radar, Global Positioning System, and so on. In the United States, the Federal Communications Commission decides who gets to use which frequencies and for what purpose.

The *bit error rate* (BER) of a wireless link is determined by the received signal power, noise due to interference caused by the receiver hardware, interference from other sources, and characteristics of the channel. Noise is typically proportional to the radio frequency bandwidth, and a key measure is the *signal-to-noise ratio* (SNR) required to achieve a given BER. Figure E.14 lists more challenges for wireless communication.

Typically, wireless communication is selected because the communicating devices are mobile or because wiring is inconvenient, which means the wireless network must rearrange itself dynamically. Such rearrangement makes routing

Challenge	Description	Impact
Path loss	Received power divided by transmitted power; the radio must overcome signal-to-noise ratio (SNR) of noise from interference. Path loss is exponential in distance and depends on interference if it is above 100 meters.	1 W transmit power, 1 GHz transmit frequency, 1 Mbit/sec data rate at 10^{-7} BER, distance between radios can be 728 meters in free space vs. 4 meters in a dense jungle.
Shadow fading	Received signal blocked by objects, buildings outdoors, or walls indoors; increase power to improve received SNR. It depends on the number of objects and their dielectric properties.	If transmitter is moving, need to change transmit power to ensure received SNR in region.
Multipath fading	Interference between multiple versions of signal that arrive at different times, determined by time between fastest signal and slowest signal relative to signal bandwidth.	900 MHz transmit frequency signal power changes every 30 cm.
Interference	Frequency reuse, adjacent channel, narrow band interference.	Requires filters, spread spectrum.

Figure E.14 Challenges for wireless communication.

more challenging. A second challenge is that wireless signals are not protected and hence are subject to mutual interference, especially as devices move. Power is another challenge for wireless communication, both because the devices tend to be battery powered and because antennas radiate power to communicate and little of it reaches the receiver. As a result, raw bit error rates are typically a thousand to a million times higher than copper wire.

There are two primary architectures for wireless networks: *base station* architectures and *peer-to-peer* architectures. Base stations are connected by landlines for longer-distance communication, and the mobile units communicate only with a single local base station. Peer-to-peer architectures allow mobile units to communicate with each other, and messages hop from one unit to the next until delivered to the desired unit. Although peer-to-peer is more reconfigurable, base stations tend to be more reliable since there is only one hop between the device and the station. *Cellular telephony*, the most popular example of wireless networks, relies on radio with base stations.

Cellular systems exploit exponential path loss to reuse the same frequency at spatially separated locations, thereby greatly increasing the number of customers served. Cellular systems will divide a city into nonoverlapping hexagonal cells that use different frequencies if nearby, reusing a frequency only when cells are far enough apart so that mutual interference is acceptable.

At the intersection of three hexagonal cells is a base station with transmitters and antennas that is connected to a switching office that coordinates handoffs when a mobile device leaves one cell and goes into another, as well as accepts and places calls over landlines. Depending on topography, population, and so on, the radius of a typical cell is 2 to 10 miles.

The Cell Phone

Figure E.15 shows the components of a radio, which is the heart of a cell phone. Radio signals are first received by the antenna, amplified, passed through a mixer, then filtered, demodulated, and finally decoded. The antenna acts as the interface between the medium through which radio waves travel and the electronics of the transmitter or receiver. Antennas can be designed to work best in particular directions, giving both transmission and reception directional properties. Modulation encodes information in the amplitude, phase, or frequency of the signal to increase its robustness under impaired conditions. Radio transmitters go through the same steps, just in the opposite order.

Originally, all components were analog, but over time most were replaced by digital components, requiring the radio signal to be converted from analog to digital. The desire for flexibility in the number of radio bands led to software routines replacing some of these functions in programmable chips, such as digital signal processors. Because such processors are typically found in mobile devices, emphasis is placed on performance per joule to extend battery life, performance per square millimeter of silicon to reduce size and cost, and bytes per task to reduce memory size.

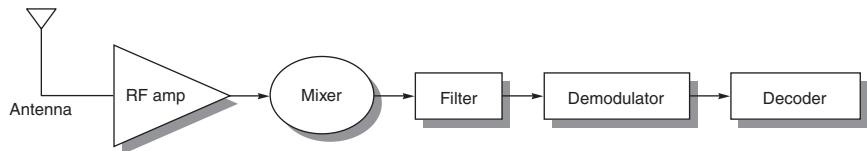


Figure E.15 A radio receiver consists of an antenna, radio frequency amplifier, mixer, filters, demodulator, and decoder. A mixer accepts two signal inputs and forms an output signal at the sum and difference frequencies. Filters select a narrower band of frequencies to pass on to the next stage. Modulation encodes information to make it more robust. Decoding turns signals into information. Depending on the application, all electrical components can be either analog or digital. For example, a car radio is all analog components, but a PC modem is all digital except for the amplifier. Today analog silicon chips are used for the RF amplifier and first mixer in cellular phones.

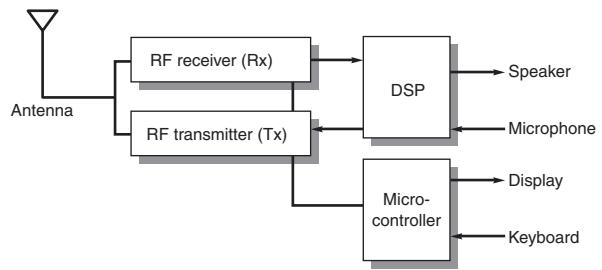


Figure E.16 Block diagram of a cell phone. The DSP performs the signal processing steps of Figure E.15, and the microcontroller controls the user interface, battery management, and call setup. (Based on Figure 1.3 of Groe and Larson [2000].)

Figure E.16 shows the generic block diagram of the electronics of a cell phone handset, with the DSP performing the signal processing and the microcontroller handling the rest of the tasks. Cell phone handsets are basically mobile computers acting as a radio. They include standard I/O devices—keyboard and LCD display—plus a microphone, speaker, and antenna for wireless networking. Battery efficiency affects sales, both for standby power when waiting for a call and for minutes of speaking.

When a cell phone is turned on, the first task is to find a cell. It scans the full bandwidth to find the strongest signal, which it keeps doing every seven seconds or if the signal strength drops, since it is designed to work from moving vehicles. It then picks an unused radio channel. The local switching office registers the cell phone and records its phone number and electronic serial number, and assigns it a voice channel for the phone conversation. To be sure the cell phone got the right channel, the base station sends a special tone on it, which the cell phone sends back to acknowledge it. The cell phone times out after 5 seconds if it doesn't hear the supervisory tone, and it starts the process all over again. The original base station makes a handoff request to the incoming base station as the signal strength drops off.

To achieve a two-way conversation over radio, frequency bands are set aside for each direction, forming a frequency pair or *channel*. The original cellular base stations transmitted at 869.04 to 893.97 MHz (called the *forward path*), and cell phones transmitted at 824.04 to 848.97 MHz (called the *reverse path*), with the frequency gap to keep them from interfering with each other. Cells might have had between 4 and 80 channels. Channels were divided into setup channels for call setup and voice channels to handle the data or voice traffic.

The communication is done digitally, just like a modem, at 9600 bits/sec. Since wireless is a lossy medium, especially from a moving vehicle, the handset sends each message five times. To preserve battery life, the original cell phones typically transmit at two signal strengths—0.6 W and 3.0 W—depending on the distance to the cell. This relatively low power not only allows smaller batteries and thus smaller cell phones, but it also aids frequency reuse, which is the key to cellular telephony.

Figure E.17 shows a circuit board from a Nokia digital phone, with the components identified. Note that the board contains two processors. A Z-80 microcontroller is responsible for controlling the functions of the board, I/O with the keyboard and display, and coordinating with the base station. The DSP handles all signal compression and decompression. In addition there are dedicated chips for analog-to-digital and digital-to-analog conversion, amplifiers, power management, and RF interfaces.

In 2001, a cell phone had about 10 integrated circuits, including parts made in exotic technologies like gallium arsenide and silicon germanium as well as standard CMOS. The economics and desire for flexibility have shrunk this to just a few chips. However, these SOCs still contain a separate microcontroller and DSP, with code implementing many of the functions just described.



Figure E.17 Circuit board from a Nokia cell phone. (Courtesy HowStuffWorks, Inc.)

Cell Phone Standards and Evolution

Improved communication speeds for cell phones were developed with multiple standards. *Code division multiple access* (CDMA), as one popular example, uses a wider radio frequency band for a path than the original cell phones, called *advanced mobile phone service* (AMPS), a mostly analog system. The wider frequency makes it more difficult to block and is called *spread spectrum*. Other standards are *time division multiple access* (TDMA) and *global system for mobile communication* (GSM). These second-generation standards—CDMA, GSM, and TDMA—are mostly digital.

The big difference for CDMA is that all callers share the same channel, which operates at a much higher rate, and it then distinguishes the different calls by encoding each one uniquely. Each CDMA phone call starts at 9600 bits/sec; it is then encoded and transmitted as equal-sized messages at 1.25 Mbits/sec. Rather than send each signal five times as in AMPS, each bit is stretched so that it takes 11 times the minimum frequency, thereby accommodating interference and yet successful transmission. The base station receives the messages, and it separates them into the separate 9600 bit/sec streams for each call.

To enhance privacy, CDMA uses pseudorandom sequences from a set of 64 predefined codes. To synchronize the handset and base station so as to pick a common pseudorandom seed, CDMA relies on a clock from the Global Positioning System, which continuously transmits an accurate time signal. By carefully selecting the codes, the shared traffic sounds like random noise to the listener. Hence, as more users share a channel there is more noise, and the signal-to-noise ratio gradually degrades. Thus, the capacity of the CDMA system is a matter of taste, depending upon the sensitivity of the listener to background noise.

In addition, CDMA uses speech compression and varies the rate of data transferred depending upon how much activity is going on in the call. Both these techniques preserve bandwidth, which allows for more calls per cell. CDMA must regulate power carefully so that signals near the cell tower do not overwhelm those from far away, with the goal of all signals reaching the tower at about the same level. The side benefit is that CDMA handsets emit less power, which both helps battery life and increases capacity when users are close to the tower.

Thus, compared to AMPS, CDMA improves the capacity of a system by up to an order of magnitude, has better call quality, has better battery life, and enhances users' privacy. After considerable commercial turmoil, there is a new third-generation standard called *International Mobile Telephony 2000* (IMT-2000), based primarily on two competing versions of CDMA and one TDMA. This standard may lead to cell phones that work anywhere in the world.

E.8

Concluding Remarks

Embedded systems are a very broad category of computing devices. This appendix has shown just some aspects of this. For example, the TI 320C55 DSP is a relatively “RISC-like” processor designed for embedded applications, with very

fine-tuned capabilities. On the other end of the spectrum, the TI 320C64x is a very high-performance, eight-issue VLIW processor for very demanding tasks. Some processors must operate on battery power alone; others have the luxury of being plugged into line current. Unifying all of these is a need to perform some level of signal processing for embedded applications. Media extensions attempt to merge DSPs with some more general-purpose processing abilities to make these processors usable for signal processing applications. We examined several case studies, including the Sony PlayStation 2, digital cameras, and cell phones. The PS2 performs detailed three-dimensional graphics, whereas a cell phone encodes and decodes signals according to elaborate communication standards. But both have system architectures that are very different from general-purpose desktop or server platforms. In general, architectural decisions that seem practical for general-purpose applications, such as multiple levels of caching or out-of-order superscalar execution, are much less desirable in embedded applications. This is due to chip area, cost, power, and real-time constraints. The programming model that these systems present places more demands on both the programmer and the compiler for extracting parallelism.

F.1	Introduction	F-2
F.2	Interconnecting Two Devices	F-6
F.3	Connecting More Than Two Devices	F-20
F.4	Network Topology	F-30
F.5	Network Routing, Arbitration, and Switching	F-44
F.6	Switch Microarchitecture	F-56
F.7	Practical Issues for Commercial Interconnection Networks	F-66
F.8	Examples of Interconnection Networks	F-73
F.9	Internetworking	F-85
F.10	Crosscutting Issues for Interconnection Networks	F-89
F.11	Fallacies and Pitfalls	F-92
F.12	Concluding Remarks	F-100
F.13	Historical Perspective and References	F-101
	References	F-109
	Exercises	F-111

F

Interconnection Networks

**Revised by Timothy M. Pinkston, University of Southern California;
José Duato, Universitat Politècnica de València, and Simula**

"The Medium is the Message" because it is the medium that shapes and controls the search and form of human associations and actions.

Marshall McLuhan
Understanding Media (1964)

The marvels—of film, radio, and television—are marvels of one-way communication, which is not communication at all.

Milton Mayer
On the Remote Possibility of Communication (1967)

The interconnection network is the heart of parallel architecture.

Chuan-Lin Wu and Tse-Yun Feng
Interconnection Networks for Parallel and Distributed Processing (1984)

Indeed, as system complexity and integration continues to increase, many designers are finding it more efficient to route packets, not wires.

Bill Dally
Principles and Practices of Interconnection Networks (2004)

F.1 Introduction

Previous chapters and appendices cover the components of a single computer but give little consideration to the interconnection of those components and how multiple computer systems are interconnected. These aspects of computer architecture have gained significant importance in recent years. In this appendix we see how to connect individual devices together into a community of communicating devices, where the term *device* is generically used to signify anything from a component or set of components within a computer to a single computer to a system of computers. Figure F.1 shows the various elements comprising this community: end nodes consisting of devices and their associated hardware and software interfaces, links from end nodes to the interconnection network, and the interconnection network. Interconnection networks are also called *networks*, *communication subnets*, or *communication subsystems*. The interconnection of multiple networks is called *internetworking*. This relies on communication standards to convert information from one kind of network to another, such as with the Internet.

There are several reasons why computer architects should devote attention to interconnection networks. In addition to providing external connectivity, networks are commonly used to interconnect the components within a single computer at many levels, including the processor microarchitecture. Networks have long been used in mainframes, but today such designs can be found in personal computers as well, given the high demand on communication bandwidth needed to enable increased computing power and storage capacity. Switched networks are replacing buses as the normal means of communication between computers, between I/O devices, between boards, between chips, and even between modules inside chips. Computer architects must understand interconnect problems and solutions in order to more effectively design and evaluate computer systems.

Interconnection networks cover a wide range of application domains, very much like memory hierarchy covers a wide range of speeds and sizes. Networks implemented within processor chips and systems tend to share characteristics much in common with processors and memory, relying more on high-speed hardware solutions and less on a flexible software stack. Networks implemented across systems tend to share much in common with storage and I/O, relying more on the operating system and software protocols than high-speed hardware—though we are seeing a convergence these days. Across the domains, performance includes latency and effective bandwidth, and queuing theory is a valuable analytical tool in evaluating performance, along with simulation techniques.

This topic is vast—portions of Figure F.1 are the subject of entire books and college courses. The goal of this appendix is to provide for the computer architect an overview of network problems and solutions. This appendix gives introductory explanations of key concepts and ideas, presents architectural implications of interconnection network technology and techniques, and provides useful references to more detailed descriptions. It also gives a common framework for evaluating all types of interconnection networks, using a single set of terms to describe the basic

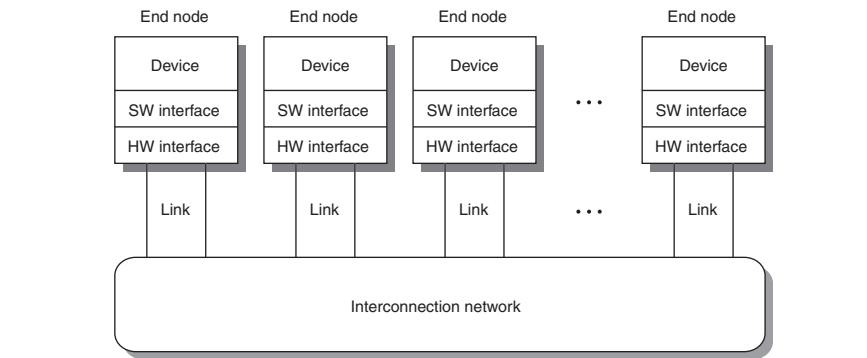


Figure F.1 A conceptual illustration of an interconnected community of devices.

alternatives. As we will see, many types of networks have common preferred alternatives, but for others the best solutions are quite different. These differences become very apparent when crossing between the networking domains.

Interconnection Network Domains

Interconnection networks are designed for use at different levels within and across computer systems to meet the operational demands of various application areas—high-performance computing, storage I/O, cluster/workgroup/enterprise systems, internetworking, and so on. Depending on the number of devices to be connected and their proximity, we can group interconnection networks into four major networking domains:

- *On-chip networks* (OCNs)—Also referred to as network-on-chip (NoC), this type of network is used for interconnecting microarchitecture functional units, register files, caches, compute tiles, and processor and IP cores within chips or multichip modules. Current and near future OCNs support the connection of a few tens to a few hundred of such devices with a maximum interconnection distance on the order of centimeters. Most OCNs used in high-performance chips are custom designed to mitigate chip-crossing wire delay problems caused by increased technology scaling and transistor integration, though some proprietary designs are gaining wider use (e.g., IBM's CoreConnect, ARM's AMBA, and Sonic's Smart Interconnect). Examples of current OCNs are those found in the Intel Teraflops processor chip [Hoskote07], connecting 80 simple cores; the Intel Single-Chip Cloud Computer (SCCC) [Howard10], connecting 48 IA-32 architecture cores; and Tilera's TILE-Gx line of processors [TILE-GX], connecting 100 processing cores in 4Q 2011 using TSMC's 40 nanometer process and 200 cores planned for 2013 (code named “Stratton”) using

TSMC’s 28 nanometer process. The networks peak at 256 GBps for both Intel prototypes and up to 200 Tbps for the TILE-Gx100 processor. More detailed information for OCNs is provided in Flich [2010].

- *System/storage area networks* (SANs)—This type of network is used for inter-processor and processor-memory interconnections within multiprocessor and multicompiler systems, and also for the connection of storage and I/O components within server and data center environments. Typically, several hundreds of such devices can be connected, although some supercomputer SANs support the interconnection of many thousands of devices, like the IBM Blue Gene/L supercomputer. The maximum interconnection distance covers a relatively small area—on the order of a few tens of meters usually—but some SANs have distances spanning a few hundred meters. For example, *InfiniBand*, a popular SAN standard introduced in late 2000, supports system and storage I/O interconnects at up to 120 Gbps over a distance of 300 m.
- *Local area networks* (LANs)—This type of network is used for interconnecting autonomous computer systems distributed across a machine room or throughout a building or campus environment. Interconnecting PCs in a cluster is a prime example. Originally, LANs connected only up to a hundred devices, but with bridging LANs can now connect up to a few thousand devices. The maximum interconnect distance covers an area of a few kilometers usually, but some have distance spans of a few tens of kilometers. For instance, the most popular and enduring LAN, *Ethernet*, has a 10 Gbps standard version that supports maximum performance over a distance of 40 km.
- *Wide area networks* (WANs)—Also called *long-haul networks*, WANs connect computer systems distributed across the globe, which requires internetworking support. WANs connect many millions of computers over distance scales of many thousands of kilometers. Asynchronous Transfer Mode (ATM) is an example of a WAN.

Figure F.2 roughly shows the relationship of these networking domains in terms of the number of devices interconnected and their distance scales. Overlap exists for some of these networks in one or both dimensions, which leads to product competition. Some network solutions have become commercial standards while others remain proprietary. Although the preferred solutions may significantly differ from one interconnection network domain to another depending on the design requirements, the problems and concepts used to address network problems remain remarkably similar across the domains. No matter the target domain, networks should be designed so as not to be the bottleneck to system performance and cost efficiency. Hence, the ultimate goal of computer architects is to design interconnection networks of the lowest possible cost that are capable of transferring the maximum amount of available information in the shortest possible time.

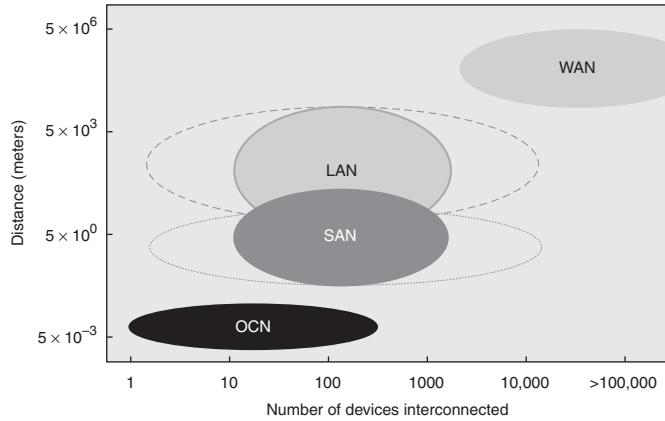


Figure F.2 Relationship of the four interconnection network domains in terms of number of devices connected and their distance scales: on-chip network (OCN), system/storage area network (SAN), local area network (LAN), and wide area network (WAN). Note that there are overlapping ranges where some of these networks compete. Some supercomputer systems use proprietary custom networks to interconnect several thousands of computers, while other systems, such as multicompiler clusters, use standard commercial networks.

Approach and Organization of This Appendix

Interconnection networks can be well understood by taking a top-down approach to unveiling the concepts and complexities involved in designing them. We do this by viewing the network initially as an opaque “black box” that simply and ideally performs certain necessary functions. Then we systematically open various layers of the black box, allowing more complex concepts and nonideal network behavior to be revealed. We begin this discussion by first considering the interconnection of just two devices in Section F.2, where the black box network can be viewed as a simple *dedicated link* network—that is, wires or collections of wires running bidirectionally between the devices. We then consider the interconnection of more than two devices in Section F.3, where the black box network can be viewed as a *shared link* network or as a *switched point-to-point* network connecting the devices. We continue to peel away various other layers of the black box by considering in more detail the network topology (Section F.4); routing, arbitration, and switching (Section F.5); and switch microarchitecture (Section F.6). Practical issues for commercial networks are considered in Section F.7, followed by examples illustrating the trade-offs for each type of network in Section F.8. Internetworking is briefly discussed in Section F.9, and additional crosscutting issues for interconnection networks are presented in Section F.10. Section F.11 gives some common fallacies

and pitfalls related to interconnection networks, and Section F.12 presents some concluding remarks. Finally, we provide a brief historical perspective and some suggested reading in Section F.13.

F.2

Interconnecting Two Devices

This section introduces the basic concepts required to understand how communication between just two networked devices takes place. This includes concepts that deal with situations in which the receiver may not be ready to process incoming data from the sender and situations in which transport errors may occur. To ease understanding, the black box network at this point can be conceptualized as an ideal network that behaves as simple dedicated links between the two devices. Figure F.3 illustrates this, where unidirectional wires run from device A to device B and *vice versa*, and each end node contains a buffer to hold the data. Regardless of the network complexity, whether dedicated link or not, a connection exists from each end node device to the network to inject and receive information to/from the network. We first describe the basic functions that must be performed at the end nodes to commence and complete communication, and then we discuss network media and the basic functions that must be performed by the network to carry out communication. Later, a simple performance model is given, along with several examples to highlight implications of key network parameters.

Network Interface Functions: Composing and Processing Messages

Suppose we want two networked devices to read a word from each other's memory. The unit of information sent or received is called a *message*. To acquire the desired data, the two devices must first compose and send a certain type of message in the form of a *request* containing the address of the data within the other device. The address (i.e., memory or operand location) allows the receiver to identify where to find the information being requested. After processing the request, each device then composes and sends another type of message, a *reply*, containing the data. The address and data information is typically referred to as the message *payload*.



Figure F.3 A simple dedicated link network bidirectionally interconnecting two devices.

In addition to payload, every message contains some control bits needed by the network to deliver the message and process it at the receiver. The most typical are bits to distinguish between different types of messages (e.g., request, reply, request acknowledge, reply acknowledge) and bits that allow the network to transport the information properly to the destination. These additional control bits are encoded in the *header* and/or *trailer* portions of the message, depending on their location relative to the message payload. As an example, Figure F.4 shows the format of a message for the simple dedicated link network shown in Figure F.3. This example shows a single-word payload, but messages in some interconnection networks can include several thousands of words.

Before message transport over the network occurs, messages have to be composed. Likewise, upon receipt from the network, they must be processed. These and other functions described below are the role of the *network interface* (also referred to as the *channel adapter*) residing at the end nodes. Together with some direct memory access (DMA) engine and link drivers to transmit/receive messages to/from the network, some dedicated memory or register(s) may be used to buffer outgoing and incoming messages. Depending on the network domain and design specifications for the network, the network interface hardware may consist of nothing more than the communicating device itself (i.e., for OCNs and some SANs) or a separate card that integrates several embedded processors and DMA engines with thousands of megabytes of RAM (i.e., for many SANs and most LANs and WANs).

In addition to hardware, network interfaces can include software or firmware to perform the needed operations. Even the simple example shown in Figure F.3 may invoke messaging software to translate requests and replies into messages with the appropriate headers. This way, user applications need not worry about composing and processing messages as these tasks can be performed automatically at a lower level. An application program usually cooperates with the operating or runtime

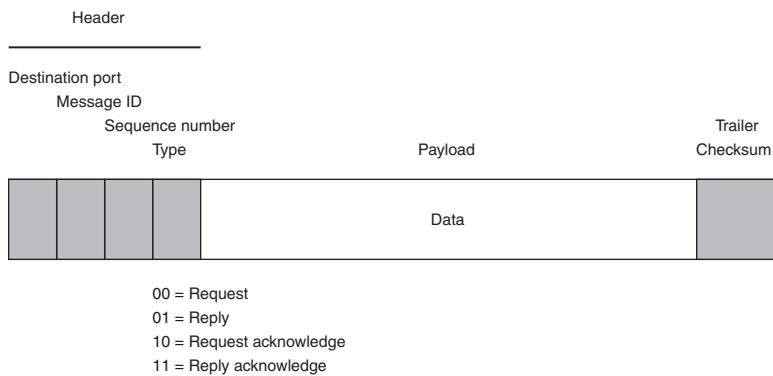


Figure F.4 An example packet format with header, payload, and checksum in the trailer.

system to send and receive messages. As the network is likely to be shared by many processes running on each device, the operating system cannot allow messages intended for one process to be received by another. Thus, the messaging software must include protection mechanisms that distinguish between processes. This distinction could be made by expanding the header with a *port* number that is known by both the sender and intended receiver processes.

In addition to composing and processing messages, additional functions need to be performed by the end nodes to establish communication among the communicating devices. Although hardware support can reduce the amount of work, some can be done by software. For example, most networks specify a maximum amount of information that can be transferred (i.e., *maximum transfer unit*) so that network buffers can be dimensioned appropriately. Messages longer than the maximum transfer unit are divided into smaller units, called *packets* (or *datagrams*), that are transported over the network. Packets are reassembled into messages at the destination end node before delivery to the application. Packets belonging to the same message can be distinguished from others by including a *message ID* field in the packet header. If packets arrive out of order at the destination, they are reordered when reassembled into a message. Another field in the packet header containing a *sequence number* is usually used for this purpose.

The sequence of steps the end node follows to commence and complete communication over the network is called a *communication protocol*. It generally has symmetric but reversed steps between sending and receiving information. Communication protocols are implemented by a combination of software and hardware to accelerate execution. For instance, many network interface cards implement hardware timers as well as hardware support to split messages into packets and reassemble them, compute the cyclic redundancy check (CRC) *checksum*, handle virtual memory addresses, and so on.

Some network interfaces include extra hardware to offload protocol processing from the host computer, such as TCP *offload engines* for LANs and WANs. But, for interconnection networks such as SANs that have low latency requirements, this may not be enough even when lighter-weight communication protocols are used such as message passing interface (MPI). Communication performance can be further improved by bypassing the operating system (OS). OS bypassing can be implemented by directly allocating message buffers in the network interface memory so that applications directly write into and read from those buffers. This avoids extra memory-to-memory copies. The corresponding protocols are referred to as *zero-copy* protocols or *user-level communication* protocols. Protection can still be maintained by calling the OS to allocate those buffers at initialization and preventing unauthorized memory accesses in hardware.

In general, some or all of the following are the steps needed to send a message at end node devices over a network:

1. The application executes a system call, which copies data to be sent into an operating system or network interface buffer, divides the message into packets (if needed), and composes the header and trailer for packets.

2. The checksum is calculated and included in the header or trailer of packets.
3. The timer is started, and the network interface hardware sends the packets.

Message reception is in the reverse order:

3. The network interface hardware receives the packets and puts them into its buffer or the operating system buffer.
2. The checksum is calculated for each packet. If the checksum matches the sender's checksum, the receiver sends an acknowledgment back to the packet sender. If not, it deletes the packet, assuming that the sender will resend the packet when the associated timer expires.
1. Once all packets pass the test, the system reassembles the message, copies the data to the user's address space, and signals the corresponding application.

The sender must still react to packet acknowledgments:

- When the sender gets an acknowledgment, it releases the copy of the corresponding packet from the buffer.
- If the sender reaches the time-out instead of receiving an acknowledgment, it resends the packet and restarts the timer.

Just as a protocol is implemented at network end nodes to support communication, protocols are also used across the network structure at the physical, data link, and network layers responsible primarily for packet transport, flow control, error handling, and other functions described next.

Basic Network Structure and Functions: Media and Form Factor, Packet Transport, Flow Control, and Error Handling

Once a packet is ready for transmission at its source, it is injected into the network using some dedicated hardware at the network interface. The hardware includes some transceiver circuits to drive the physical network media—either electrical or optical. The type of *media* and *form factor* depends largely on the interconnect distances over which certain signaling rates (e.g., transmission speed) should be sustainable. For centimeter or less distances on a chip or multichip module, typically the middle to upper copper metal layers can be used for interconnects at multi-Gbps signaling rates per line. A dozen or more layers of copper traces or tracks imprinted on circuit boards, midplanes, and backplanes can be used for Gbps differential-pair signaling rates at distances of about a meter or so. Category 5E unshielded twisted-pair copper wiring allows 0.25 Gbps transmission speed over distances of 100 meters. Coaxial copper cables can deliver 10 Mbps over kilometer distances. In these conductor lines, distance can usually be traded off for higher transmission speed, up to a certain point. Optical media enable faster transmission

speeds at distances of kilometers. Multimode fiber supports 100 Mbps transmission rates over a few kilometers, and more expensive single-mode fiber supports Gbps transmission speeds over distances of several kilometers. Wavelength division multiplexing allows several times more bandwidth to be achieved in fiber (i.e., by a factor of the number of wavelengths used).

The hardware used to drive network links may also include some encoders to encode the signal in a format other than binary that is suitable for the given transport distance. Encoding techniques can use multiple voltage levels, redundancy, data and control rotation (e.g., 4b5b encoding), and/or a guaranteed minimum number of signal transitions per unit time to allow for clock recovery at the receiver. The signal is decoded at the receiver end, and the packet is stored in the corresponding buffer. All of these operations are performed at the network physical layer, the details of which are beyond the scope of this appendix. Fortunately, we do not need to worry about them. From the perspective of the data link and higher layers, the physical layer can be viewed as a long linear pipeline without staging in which signals propagate as waves through the network transmission medium. All of the above functions are generally referred to as *packet transport*.

Besides packet transport, the network hardware and software are jointly responsible at the data link and network protocol layers for ensuring reliable delivery of packets. These responsibilities include: (1) preventing the sender from sending packets at a faster rate than they can be processed by the receiver, and (2) ensuring that the packet is neither garbled nor lost in transit. The first responsibility is met by either discarding packets at the receiver when its buffer is full and later notifying the sender to retransmit them, or by notifying the sender to stop sending packets when the buffer becomes full and to resume later once it has room for more packets. The latter strategy is generally known as *flow control*.

There are several interesting techniques commonly used to implement flow control beyond simple *handshaking* between the sender and receiver. The more popular techniques are *Xon/Xoff* (also referred to as *Stop & Go*) and *credit-based* flow control. Xon/Xoff consists of the receiver notifying the sender either to stop or to resume sending packets once high and low buffer occupancy levels are reached, respectively, with some hysteresis to reduce the number of notifications. Notifications are sent as “stop” and “go” signals using additional control wires or encoded in control packets. Credit-based flow control typically uses a credit counter at the sender that initially contains a number of credits equal to the number of buffers at the receiver. Every time a packet is transmitted, the sender decrements the credit counter. When the receiver consumes a packet from its buffer, it returns a credit to the sender in the form of a control packet that notifies the sender to increment its counter upon receipt of the credit. These techniques essentially control the flow of packets into the network by *throttling* packet injection at the sender when the receiver reaches a low watermark or when the sender runs out of credits.

Xon/Xoff usually generates much less control traffic than credit-based flow control because notifications are only sent when the high or low buffer occupancy levels are crossed. On the other hand, credit-based flow control requires less than half the buffer size required by Xon/Xoff. Buffers for Xon/Xoff must be large

enough to prevent overflow before the “stop” control signal reaches the sender. Overflow cannot happen when using credit-based flow control because the sender will run out of credits, thus stopping transmission. For both schemes, full link bandwidth utilization is possible only if buffers are large enough for the distance over which communication takes place.

Let’s compare the buffering requirements of the two flow control techniques in a simple example covering the various interconnection network domains.

Example Suppose we have a dedicated-link network with a raw data bandwidth of 8 Gbps for each link in each direction interconnecting two devices. Packets of 100 bytes (including the header) are continuously transmitted from one device to the other to fully utilize network bandwidth. What is the minimum amount of credits and buffer space required by credit-based flow control assuming interconnect distances of 1 cm, 1 m, 100 m, and 10 km if only link propagation delay is taken into account? How does the minimum buffer space compare against Xon/Xoff?

Answer At the start, the receiver buffer is initially empty and the sender contains a number of credits equal to buffer capacity. The sender will consume a credit every time a packet is transmitted. For the sender to continue transmitting packets at network speed, the first returned credit must reach the sender before the sender runs out of credits. After receiving the first credit, the sender will keep receiving credits at the same rate it transmits packets. As we are considering only propagation delay over the link and no other sources of delay or overhead, null processing time at the sender and receiver are assumed. The time required for the first credit to reach the sender since it started transmission of the first packet is equal to the round-trip propagation delay for the packet transmitted to the receiver and the return credit transmitted back to the sender. This time must be less than or equal to the packet transmission time multiplied by the initial credit count:

$$\text{Packet propagation delay} + \text{Credit propagation delay} \leq \frac{\text{Packet size}}{\text{Bandwidth}} \times \text{Credit count}$$

The speed of light is about 300,000 km/sec. Assume we can achieve 66% of that in a conductor. Thus, the minimum number of credits for each distance is given by

$$\left(\frac{\text{Distance}}{2/3 \times 300,000 \text{ km/sec}} \right) \times 2 \leq \frac{100 \text{ bytes}}{8 \text{ Gbits/sec}} \times \text{Credit count}$$

As each credit represents one packet-sized buffer entry, the minimum amount of credits (and, likewise, buffer space) needed by each device is one for the 1 cm and 1 m distances, 10 for the 100 m distance, and 1000 packets for the 10 km distance. For Xon/Xoff, this minimum buffer size corresponds to the buffer fragment from the high occupancy level to the top of the buffer and from the low occupancy level to the bottom of the buffer. With the added hysteresis between both occupancy levels to reduce notifications, the minimum buffer space for Xon/Xoff turns out to be more than twice that for credit-based flow control.

Networks that implement flow control do not need to drop packets and are sometimes referred to as *lossless* networks; networks that drop packets are sometimes referred to as *lossy* networks. This single difference in the way packets are handled by the network drastically constrains the kinds of solutions that can be implemented to address other related network problems, including packet routing, congestion, deadlock, and reliability, as we will see later in this appendix. This difference also affects performance significantly as dropped packets need to be retransmitted, thus consuming more link bandwidth and suffering extra delay. These behavioral and performance differences ultimately restrict the interconnection network domains for which certain solutions are applicable. For instance, most networks delivering packets over relatively short distances (e.g., OCNs and SANs) tend to implement flow control; on the other hand, networks delivering packets over relatively long distances (e.g., LANs and WANs) tend to be designed to drop packets. For the shorter distances, the delay in propagating flow control information back to the sender can be negligible, but not so for longer distance scales. The kinds of applications that are usually run also influence the choice of lossless versus lossy networks. For instance, dropping packets sent by an Internet client like a Web browser affects only the delay observed by the corresponding user. However, dropping a packet sent by a process from a parallel application may lead to a significant increase in the overall execution time of the application if that packet's delay is on the critical path.

The second responsibility of ensuring that packets are neither garbled nor lost in transit can be met by implementing some mechanisms to detect and recover from transport errors. Adding a checksum or some other error detection field to the packet format, as shown in Figure F.4, allows the receiver to detect errors. This redundant information is calculated when the packet is sent and checked upon receipt. The receiver then sends an acknowledgment in the form of a control packet if the packet passes the test. Note that this acknowledgment control packet may simultaneously contain flow control information (e.g., a credit or stop signal), thus reducing control packet overhead. As described earlier, the most common way to recover from errors is to have a timer record the time each packet is sent and to presume the packet is lost or erroneously transported if the timer expires before an acknowledgment arrives. The packet is then resent.

The communication protocol across the network and network end nodes must handle many more issues other than packet transport, flow control, and reliability. For example, if two devices are from different manufacturers, they might order bytes differently within a word (Big Endian versus Little Endian byte ordering). The protocol must reverse the order of bytes in each word as part of the delivery system. It must also guard against the possibility of duplicate packets if a delayed packet were to become unstuck. Depending on the system requirements, the protocol may have to implement *pipelining* among operations to improve performance. Finally, the protocol may need to handle network congestion to prevent performance degradation when more than two devices are connected, as described later in Section F.7.

Characterizing Performance: Latency and Effective Bandwidth

Now that we have covered the basic steps for sending and receiving messages between two devices, we can discuss performance. We start by discussing the latency when transporting a single packet. Then we discuss the effective bandwidth (also known as throughput) that can be achieved when the transmission of multiple packets is pipelined over the network at the packet level.

Figure F.5 shows the basic components of latency for a single packet. Note that some latency components will be broken down further in later sections as the internals of the “black box” network are revealed. The timing parameters in Figure F.5 apply to many interconnection network domains: inside a chip, between chips on a board, between boards in a chassis, between chassis within a computer, between computers in a cluster, between clusters, and so on. The values may change, but the components of latency remain the same.

The following terms are often used loosely, leading to confusion, so we define them here more precisely:

- **Bandwidth**—Strictly speaking, the *bandwidth* of a transmission medium refers to the range of frequencies for which the attenuation per unit length introduced by that medium is below a certain threshold. It must be distinguished from the *transmission speed*, which is the amount of information transmitted over a medium per unit time. For example, modems successfully increased transmission speed in the late 1990s for a fixed bandwidth (i.e., the 3 KHz bandwidth provided by voice channels over telephone lines) by encoding more voltage levels and, hence, more bits per signal cycle. However, to be consistent with

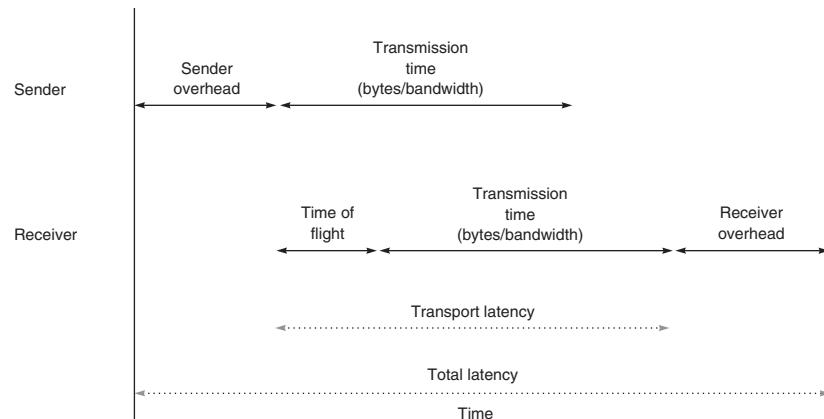


Figure F.5 Components of packet latency. Depending on whether it is an OCN, SAN, LAN, or WAN, the relative amounts of sending and receiving overhead, time of flight, and transmission time are usually quite different from those illustrated here.

its more widely understood meaning, we use the term *band-width* to refer to the maximum rate at which information can be transferred, where information includes packet header, payload, and trailer. The units are traditionally bits per second, although bytes per second is sometimes used. The term *bandwidth* is also used to mean the measured speed of the medium (i.e., network links). *Aggregate bandwidth* refers to the total data bandwidth supplied by the network, and *effective bandwidth* or *throughput* is the fraction of aggregate bandwidth delivered by the network to an application.

- *Time of flight*—This is the time for the first bit of the packet to arrive at the receiver, including the propagation delay over the links and delays due to other hardware in the network such as link repeaters and network switches. The unit of measure for time of flight can be in milliseconds for WANs, microseconds for LANs, nanoseconds for SANs, and picoseconds for OCNs.
- *Transmission time*—This is the time for the packet to pass through the network, not including time of flight. One way to measure it is the difference in time between when the first bit of the packet arrives at the receiver and when the last bit of that packet arrives at the receiver. By definition, transmission time is equal to the size of the packet divided by the data bandwidth of network links. This measure assumes there are no other packets contending for that bandwidth (i.e., a zero-load or no-load network).
- *Transport latency*—This is the sum of time of flight and transmission time. Transport latency is the time that the packet spends in the interconnection network. Stated alternatively, it is the time between when the first bit of the packet is injected into the network and when the last bit of that packet arrives at the receiver. It does not include the overhead of preparing the packet at the sender or processing it when it arrives at the receiver.
- *Sending overhead*—This is the time for the end node to prepare the packet (as opposed to the message) for injection into the network, including both hardware and software components. Note that the end node is busy for the entire time, hence the use of the term *overhead*. Once the end node is free, any subsequent delays are considered part of the transport latency. We assume that overhead consists of a constant term plus a variable term that depends on packet size. The constant term includes memory allocation, packet header preparation, setting up DMA devices, and so on. The variable term is mostly due to copies from buffer to buffer and is usually negligible for very short packets.
- *Receiving overhead*—This is the time for the end node to process an incoming packet, including both hardware and software components. We also assume here that overhead consists of a constant term plus a variable term that depends on packet size. In general, the receiving overhead is larger than the sending overhead. For example, the receiver may pay the cost of an interrupt or may have to reorder and reassemble packets into messages.

The total latency of a packet can be expressed algebraically by the following:

$$\text{Latency} = \text{Sending overhead} + \text{Time of flight} + \frac{\text{Packet size}}{\text{Bandwidth}} + \text{Receiving overhead}$$

Let's see how the various components of transport latency and the sending and receiving overheads change in importance as we go across the interconnection network domains: from OCNs to SANs to LANs to WANs.

Example Assume that we have a dedicated link network with a data bandwidth of 8 Gbps for each link in each direction interconnecting two devices within an OCN, SAN, LAN, or WAN, and we wish to transmit packets of 100 bytes (including the header) between the devices. The end nodes have a per-packet sending overhead of $x + 0.05$ ns/byte and receiving overhead of $4/3(x) + 0.05$ ns/byte, where x is 0 μ s for the OCN, 0.3 μ s for the SAN, 3 μ s for the LAN, and 30 μ s for the WAN, which are typical for these network types. Calculate the total latency to send packets from one device to the other for interconnection distances of 0.5 cm, 5 m, 5000 m, and 5000 km assuming that time of flight consists only of link propagation delay (i.e., no switching or other sources of delay).

Answer Using the above expression and the calculation for propagation delay through a conductor given in the previous example, we can plug in the parameters for each of the networks to find their total packet latency. For the OCN:

$$\begin{aligned}\text{Latency} &= \text{Sending overhead} + \text{Time of flight} + \frac{\text{Packet size}}{\text{Bandwidth}} + \text{Receiving overhead} \\ &= 5 \text{ ns} + \frac{0.5 \text{ cm}}{2/3 \times 300,000 \text{ km/sec}} + \frac{100 \text{ bytes}}{8 \text{ Gbits/sec}} + 5 \text{ ns}\end{aligned}$$

Converting all terms into nanoseconds (ns) leads to the following for the OCN:

$$\begin{aligned}\text{Total latency (OCN)} &= 5 \text{ ns} + \frac{0.5 \text{ cm}}{2/3 \times 300,000 \text{ km/sec}} + \frac{100 \times 8}{8} \text{ ns} + 5 \text{ ns} \\ &= 5 \text{ ns} + 0.025 \text{ ns} + 100 \text{ ns} + 5 \text{ ns} \\ &= 110.025 \text{ ns}\end{aligned}$$

Substituting in the appropriate values for the SAN gives the following latency:

$$\begin{aligned}\text{Total latency (SAN)} &= 0.305 \mu\text{s} + \frac{5 \text{ m}}{2/3 \times 300,000 \text{ km/sec}} + \frac{100 \text{ bytes}}{8 \text{ Gbits/sec}} + 0.405 \mu\text{s} \\ &= 0.305 \mu\text{s} + 0.025 \mu\text{s} + 0.1 \mu\text{s} + 0.405 \mu\text{s} \\ &= 0.835 \mu\text{s}\end{aligned}$$

Substituting in the appropriate values for the LAN gives the following latency:

$$\begin{aligned}\text{Total latency (LAN)} &= 3.005 \mu\text{s} + \frac{5 \text{ km}}{2/3 \times 300,000 \text{ km/sec}} + \frac{100 \text{ bytes}}{8 \text{ Gbits/sec}} + 4.005 \mu\text{s} \\ &= 3.005 \mu\text{s} + 25 \mu\text{s} + 0.1 \mu\text{s} + 4.005 \mu\text{s} \\ &= 32.11 \mu\text{s}\end{aligned}$$

Substituting in the appropriate values for the WAN gives the following latency:

$$\begin{aligned}\text{Total latency (WAN)} &= 30.005 \mu\text{s} + \frac{5000 \text{ km}}{2/3 \times 300,000 \text{ km/sec}} + \frac{100 \text{ bytes}}{8 \text{ Gbits/sec}} + 40.005 \mu\text{s} \\ &= 30.005 \mu\text{s} + 25000 \mu\text{s} + 0.1 \mu\text{s} + 40.005 \mu\text{s} \\ &= 25.07 \text{ ms}\end{aligned}$$

The increased fraction of the latency required by time of flight for the longer distances along with the greater likelihood of errors over the longer distances are among the reasons why WANs and LANs use more sophisticated and time-consuming communication protocols, which increase sending and receiving overheads. The need for standardization is another reason. Complexity also increases due to the requirements imposed on the protocol by the typical applications that run over the various interconnection network domains as we go from tens to hundreds to thousands to many thousands of devices. We will consider this in later sections when we discuss connecting more than two devices. The above example shows that the propagation delay component of time of flight for WANs and some LANs is so long that other latency components—including the sending and receiving overheads—can practically be ignored. This is not so for SANs and OCNs where the propagation delay pales in comparison to the overheads and transmission delay. Remember that time-of-flight latency due to switches and other hardware in the network besides sheer propagation delay through the links is neglected in the above example. For noncongested networks, switch latency generally is small compared to the overheads and propagation delay through the links in WANs and LANs, but this is not necessarily so for multiprocessor SANs and multicore OCNs, as we will see in later sections.

So far, we have considered the transport of a single packet and computed the associated end-to-end total packet latency. In order to compute the effective bandwidth for two networked devices, we have to consider a continuous stream of packets transported between them. We must keep in mind that, in addition to minimizing packet latency, the goal of any network optimized for a given cost and power consumption target is to transfer the maximum amount of available information in the shortest possible time, as measured by the effective bandwidth delivered by the network. For applications that do not require a response before sending the next packet, the sender can overlap the sending overhead of later packets with the transport latency and receiver overhead of prior packets. This essentially pipelines the transmission of packets over the network, also known as *link pipelining*. Fortunately, as discussed in prior chapters of this book, there are many application

areas where communication from either several applications or several threads from the same application can run concurrently (e.g., a Web server concurrently serving thousands of client requests or streaming media), thus allowing a device to send a stream of packets without having to wait for an acknowledgment or a reply. Also, as long messages are usually divided into packets of maximum size before transport, a number of packets are injected into the network in succession for such cases. If such overlap were not possible, packets would have to wait for prior packets to be acknowledged before being transmitted and thus suffer significant performance degradation.

Packets transported in a pipelined fashion can be acknowledged quite straightforwardly simply by keeping a copy at the source of all unacknowledged packets that have been sent and keeping track of the correspondence between returned acknowledgments and packets stored in the buffer. Packets will be removed from the buffer when the corresponding acknowledgment is received by the sender. This can be done by including the message ID and packet sequence number associated with the packet in the packet's acknowledgment. Furthermore, a separate timer must be associated with each buffered packet, allowing the packet to be resent if the associated time-out expires.

Pipelining packet transport over the network has many similarities with pipelining computation within a processor. However, among some differences are that it does not require any staging latches. Information is simply propagated through network links as a sequence of signal waves. Thus, the network can be considered as a logical pipeline consisting of as many stages as are required so that the time of flight does not affect the effective bandwidth that can be achieved. Transmission of a packet can start immediately after the transmission of the previous one, thus overlapping the sending overhead of a packet with the transport and receiver latency of previous packets. If the sending overhead is smaller than the transmission time, packets follow each other back-to-back, and the effective bandwidth approaches the raw link bandwidth when continuously transmitting packets. On the other hand, if the sending overhead is greater than the transmission time, the effective bandwidth at the injection point will remain well below the raw link bandwidth. The resulting *link injection bandwidth*, $BW_{LinkInjection}$, for each link injecting a continuous stream of packets into a network is calculated with the following expression:

$$BW_{LinkInjection} = \frac{\text{Packet size}}{\max(\text{Sending overhead}, \text{Transmission time})}$$

We must also consider what happens if the receiver is unable to consume packets at the same rate they arrive. This occurs if the receiving overhead is greater than the sending overhead and the receiver cannot process incoming packets fast enough. In this case, the *link reception bandwidth*, $BW_{LinkReception}$, for each reception link of the network is less than the link injection bandwidth and is obtained with this expression:

$$BW_{LinkReception} = \frac{\text{Packet size}}{\max(\text{Receiving overhead}, \text{Transmission time})}$$

When communication takes place between two devices interconnected by dedicated links, all the packets sent by one device will be received by the other. If the receiver cannot process packets fast enough, the receiver buffer will become full, and flow control will throttle transmission at the sender. As this situation is produced by causes external to the network, we will not consider it further here. Moreover, if the receiving overhead is greater than the sending overhead, the receiver buffer will fill up and flow control will, likewise, throttle transmission at the sender. In this case, the effect of flow control is, on average, the same as if we replace sending overhead with receiving overhead. Assuming an ideal network that behaves like two dedicated links running in opposite directions at the full link bandwidth between the two devices—which is consistent with our black box view of the network to this point—the resulting effective bandwidth is the smaller of twice the injection bandwidth (to account for the two injection links, one for each device) or twice the reception bandwidth. This results in the following expression for effective bandwidth:

$$\text{Effective bandwidth} = \min(2 \times \text{BW}_{\text{LinkInjection}}, 2 \times \text{BW}_{\text{LinkReception}}) = \frac{2 \times \text{Packet size}}{\max(\text{Overhead}, \text{Transmission time})}$$

where Overhead = max(Sending overhead, Receiving overhead). Taking into account the expression for the transmission time, it is obvious that the effective bandwidth delivered by the network is identical to the aggregate network bandwidth when the transmission time is greater than the overhead. Therefore, full network utilization is achieved regardless of the value for the time of flight and, thus, regardless of the distance traveled by packets, assuming ideal network behavior (i.e., enough credits and buffers are provided for credit-based and Xon/Xoff flow control). This analysis assumes that the sender and receiver network interfaces can process only one packet at a time. If multiple packets can be processed in parallel (e.g., as is done in IBM's Federation network interfaces), the overheads for those packets can be overlapped, which increases effective bandwidth by that overlap factor up to the amount bounded by the transmission time.

Let's use the equation on page F-17 to explore the impact of packet size, transmission time, and overhead on $\text{BW}_{\text{Link Injection}}$, $\text{BW}_{\text{LinkReception}}$, and effective bandwidth for the various network domains: OCNs, SANs, LANs, and WANs.

Example

As in the previous example, assume we have a dedicated link network with a data bandwidth of 8 Gbps for each link in each direction interconnecting the two devices within an OCN, SAN, LAN, or WAN. Plot effective bandwidth versus packet size for each type of network for packets ranging in size from 4 bytes (i.e., a single 32-bit word) to 1500 bytes (i.e., the maximum transfer unit for Ethernet), assuming that end nodes have the same per-packet sending and receiving overheads as before: $x + 0.05 \text{ ns}/\text{byte}$ and $4/3(x) + 0.05 \text{ ns}/\text{byte}$, respectively, where x is 0 μs for the OCN, 0.3 μs for the SAN, 3 μs for the LAN, and 30 μs for the WAN. What limits the effective bandwidth, and for what packet sizes is the effective bandwidth within 10% of the aggregate network bandwidth?

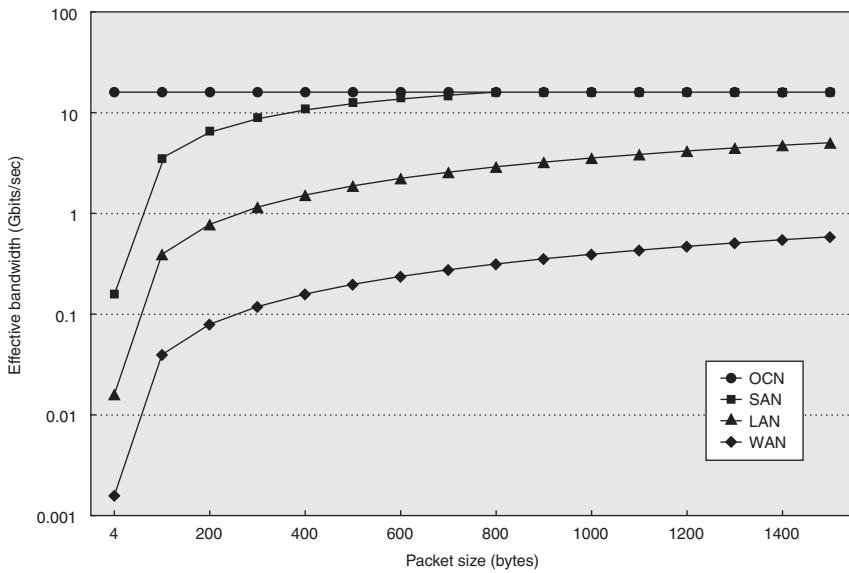


Figure F.6 Effective bandwidth versus packet size plotted in semi-log form for the four network domains. Overhead can be amortized by increasing the packet size, but for too large of an overhead (e.g., for WANs and some LANs) scaling the packet size is of little help. Other considerations come into play that limit the maximum packet size.

Answer Figure F.6 plots effective bandwidth versus packet size for the four network domains using the simple equation and parameters given above. For all packet sizes in the OCN, transmission time is greater than overhead (sending or receiving), allowing full utilization of the aggregate bandwidth, which is 16 Gbps—that is, injection link (alternatively, reception link) bandwidth times two to account for both devices. For the SAN, overhead—specifically, receiving overhead—is larger than transmission time for packets less than about 800 bytes; consequently, packets of 655 bytes and larger are needed to utilize 90% or more of the aggregate bandwidth. For LANs and WANs, most of the link bandwidth is not utilized since overhead in this example is many times larger than transmission time for all packet sizes.

This example highlights the importance of reducing the sending and receiving overheads relative to packet transmission time in order to maximize the effective bandwidth delivered by the network.

The analysis above suggests that it is possible to provide some upper bound for the effective bandwidth by analyzing the path followed by packets and determining where the bottleneck occurs. We can extend this idea beyond the network interfaces by defining a model that considers the entire network from end to

end as a pipe and identifying the narrowest section of that pipe. There are three areas of interest in that pipe: the aggregate of all network injection links and the corresponding *network injection bandwidth* ($BW_{NetworkInjection}$), the aggregate of all network reception links and the corresponding *network reception bandwidth* ($BW_{NetworkReception}$), and the aggregate of all network links and the corresponding *network bandwidth* ($BW_{Network}$). Expressions for these will be given in later sections as various layers of the black box view of the network are peeled away.

To this point, we have assumed that for just two interconnected devices the black box network behaves ideally and the network bandwidth is equal to the aggregate raw network bandwidth. In reality, it can be much less than the aggregate bandwidth as we will see in the following sections. In general, the effective bandwidth delivered end-to-end by the network to an application is upper bounded by the minimum across all three potential bottleneck areas:

$$\text{Effective bandwidth} = \min(BW_{NetworkInjection}, BW_{Network}, BW_{NetworkReception})$$

We will expand upon this expression further in the following sections as we reveal more about interconnection networks and consider the more general case of interconnecting more than two devices.

In some sections of this appendix, we show how the concepts introduced in the section take shape in example high-end commercial products. Figure F.7 lists several commercial computers that, at one point in time in their existence, were among the highest-performing systems in the world within their class. Although these systems are capable of interconnecting more than two devices, they implement the basic functions needed for interconnecting only two devices. In addition to being applicable to the SANs used in those systems, the issues discussed in this section also apply to other interconnect domains: from OCNs to WANs.

F.3

Connecting More than Two Devices

To this point, we have considered the connection of only two devices communicating over a network viewed as a black box, but what makes interconnection networks interesting is the ability to connect hundreds or even many thousands of devices together. Consequently, what makes them interesting also makes them more challenging to build. In order to connect more than two devices, a suitable structure and more functionality must be supported by the network. This section continues with our black box approach by introducing, at a conceptual level, additional network structure and functions that must be supported when interconnecting more than two devices. More details on these individual subjects are given in Sections F.4 through F.7. Where applicable, we relate the additional structure and functions to network media, flow control, and other basics presented in the previous section. In this section, we also classify networks into two broad categories

Company	System [network] name	Intro year	Max. number of compute nodes [<# CPUs>]	System footprint for max. configuration	Packet [header] max size (bytes)	Injection [reception] node BW in MB/sec	Minimum send/receive overhead	Maximum copper link length; flow control; error
Intel	ASCI Red Paragon	2001	4510 [$\times 2$]	2500 ft ²	1984 [4]	400 [400]	Few μ s	Handshaking; CRC + parity
IBM	ASCI White SP Power3 [Colony]	2001	512 [$\times 16$]	10,000 ft ²	1024 [6]	500 [500]	$\sim 3 \mu$ s	25 m; credit-based; CRC
Intel	Thunder Itanium2 Tiger4 [QsNet ^{II}]	2004	1024 [$\times 4$]	120 m ²	2048 [14]	928 [928]	0.240 μ s	13 m; credit-based; CRC for link, dest.
Cray	XT3 [SeaStar]	2004	30,508 [$\times 1$]	263.8 m ²	80 [16]	3200 [3200]	Few μ s	7 m; credit-based; CRC
Cray	X1E	2004	1024 [$\times 1$]	27 m ²	32 [16]	1600 [1600]	0 (direct LD ST accesses)	5 m; credit-based; CRC
IBM	ASC Purple pSeries 575 [Federation]	2005	>1280 [$\times 8$]	6720 ft ²	2048 [7]	2000 [2000]	$\sim 1 \mu$ s with up to 4 packets processed in	25 m; credit-based; CRC
IBM	Blue Gene/L eServer Sol. [Torus Net.]	2005	65,536 [$\times 2$]	2500 ft ² (.9 \times .9 \times 1.9 m ³ / 1 K node rack)	256 [8]	612.5 [1050]	$\sim 3 \mu$ s (2300 cycles)	8.6 m; credit-based; CRC (header/pkt)

Figure F.7 Basic characteristics of interconnection networks in commercial high-performance computer systems.

based on their connection structure—*shared-media* versus *switched-media* networks—and we compare them. Finally, expanded expressions for characterizing network performance are given, followed by an example.

Additional Network Structure and Functions: Topology, Routing, Arbitration, and Switching

Networks interconnecting more than two devices require mechanisms to physically connect the packet source to its destination in order to transport the packet and deliver it to the correct destination. These mechanisms can be implemented in different ways and significantly vary across interconnection network domains. However, the types of network structure and functions performed by those mechanisms are very much the same, regardless of the domain.

When multiple devices are interconnected by a network, the connections between them oftentimes cannot be permanently established with dedicated links.

This could either be too restrictive as all the packets from a given source would go to the same one destination (and not to others) or prohibitively expensive as a dedicated link would be needed from every source to every destination (we will evaluate this further in the next section). Therefore, networks usually share paths among different pairs of devices, but how those paths are shared is determined by the network connection structure, commonly referred to as the network *topology*. Topology addresses the important issue of “*What paths are possible for packets?*” so packets reach their intended destinations.

Every network that interconnects more than two devices also requires some mechanism to deliver each packet to the correct destination. The associated function is referred to as *routing*, which can be defined as the set of operations that need to be performed to compute a valid path from the packet source to its destinations. Routing addresses the important issue of “*Which of the possible paths are allowable (valid) for packets?*” so packets reach their intended destinations. Depending on the network, this function may be executed at the packet source to compute the entire path, at some intermediate devices to compute fragments of the path on the fly, or even at every possible destination device to verify whether that device is the intended destination for the packet. Usually, the packet header shown in Figure F.4 is extended to include the necessary routing information.

In general, as networks usually contain shared paths or parts thereof among different pairs of devices, packets may request some shared resources. When several packets request the same resources at the same time, an *arbitration* function is required to resolve the conflict. Arbitration, along with flow control, addresses the important issue of “*When are paths available for packets?*” Every time arbitration is performed, there is a winner and possibly several losers. The losers are not granted access to the requested resources and are typically buffered. As indicated in the previous section, flow control may be implemented to prevent buffer overflow. The winner proceeds toward its destination once the granted resources are switched in, providing a path for the packet to advance. This function is referred to as *switching*. Switching addresses the important issue of “*How are paths allocated to packets?*” To achieve better utilization of existing communication resources, most networks do not establish an entire end-to-end path at once. Instead, as explained in Section F.5, paths are usually established one fragment at a time.

These three network functions—routing, arbitration, and switching—must be implemented in every network connecting more than two devices, no matter what form the network topology takes. This is in addition to the basic functions mentioned in the previous section. However, the complexity of these functions and the order in which they are performed depends on the category of network topology, as discussed below. In general, routing, arbitration, and switching are required to establish a valid path from source to destination from among the possible paths provided by the network topology. Once the path has been established, the packet transport functions previously described are used to reliably transmit packets and receive them at the corresponding destination. Flow control, if implemented, prevents buffer overflow by throttling the sender. It can be implemented at the end-to-end level, the link level within the network, or both.

Shared-Media Networks

The simplest way to connect multiple devices is to have them share the network media, as shown for the bus in Figure F.8 (a). This has been the traditional way of interconnecting devices. The shared media can operate in *half-duplex* mode, where data can be carried in either direction over the media but simultaneous transmission and reception of data by the same device is not allowed, or in *full-duplex*, where the data can be carried in both directions and simultaneously transmitted and received by the same device. Until very recently, I/O devices in most systems typically shared a single I/O bus, and early system-on-chip (SoC) designs made use of a shared bus to interconnect on-chip components. The most popular LAN, Ethernet, was originally implemented as a half-duplex bus shared by up to a hundred computers, although now switched-media versions also exist.

Given that network media are shared, there must be a mechanism to coordinate and arbitrate the use of the shared media so that only one packet is sent at a time. If the physical distance between network devices is small, it may be possible to have a central arbiter to grant permission to send packets. In this case, the network nodes may use dedicated control lines to interface with the arbiter. Centralized arbitration is impractical, however, for networks with a large number of nodes spread over large distances, so distributed forms of arbitration are also used. This is the case for the original Ethernet shared-media LAN.

A first step toward distributed arbitration of shared media is “looking before you leap.” A node first checks the network to avoid trying to send a packet while another packet is already in the network. Listening before transmission to avoid collisions is called *carrier sensing*. If the interconnection is idle, the node tries to send. Looking first is not a guarantee of success, of course, as some other node may also decide to send at the same instant. When two nodes send at the same time,

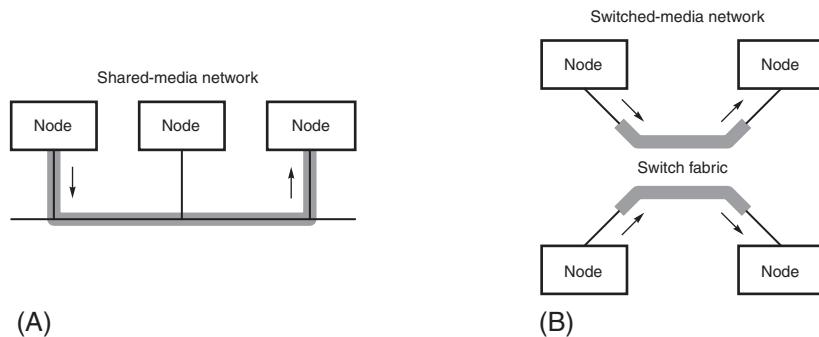


Figure F.8 (a) A shared-media network versus (b) a switched-media network. Ethernet was originally a shared media network, but switched Ethernet is now available. All nodes on the shared-media networks must dynamically share the raw bandwidth of one link, but switched-media networks can support multiple links, providing higher raw aggregate bandwidth.

a *collision* occurs. Let's assume that the network interface can detect any resulting collisions by listening to hear if the data become garbled by other data appearing on the line. Listening to detect collisions is called *collision detection*. This is the second step of distributed arbitration.

The problem is not solved yet. If, after detecting a collision, every node on the network waited exactly the same amount of time, listened to be sure there was no traffic, and then tried to send again, we could still have synchronized nodes that would repeatedly bump heads. To avoid repeated head-on collisions, each node whose packet gets garbled waits (or *backs off*) a random amount of time before resending. Randomization breaks the synchronization. Subsequent collisions result in exponentially increasing time between attempts to retransmit, so as not to tax the network.

Although this approach controls congestion on the shared media, it is not guaranteed to be fair—some subsequent node may transmit while those that collided are waiting. If the network does not have high demand from many nodes, this simple approach works well. Under high utilization, however, performance degrades since the media are shared and fairness is not ensured. Another distributed approach to arbitration of shared media that can support fairness is to pass a token between nodes. The function of the token is to grant the acquiring node the right to use the network. If the token circulates in a cyclic fashion between the nodes, a certain amount of fairness is ensured in the arbitration process.

Once arbitration has been performed and a device has been granted access to the shared media, the function of switching is straightforward. The granted device simply needs to connect itself to the shared media, thus establishing a path to every possible destination. Also, routing is very simple to implement. Given that the media are shared and attached to all the devices, every device will see every packet. Therefore, each device just needs to check whether or not a given packet is intended for that device. A beneficial side effect of this strategy is that a device can send a packet to all the devices attached to the shared media through a single transmission. This style of communication is called *broadcasting*, in contrast to *unicasting*, in which each packet is intended for only one device. The shared media make it easy to broadcast a packet to every device or, alternatively, to a subset of devices, called *multicasting*.

Switched-Media Networks

The alternative to sharing the entire network media at once across all attached nodes is to switch between disjoint portions of it shared by the nodes. Those portions consist of passive *point-to-point links* between active *switch* components that dynamically establish communication between sets of source-destination pairs. These passive and active components make up what is referred to as the network *switch fabric* or *network fabric*, to which end nodes are connected. This approach is shown conceptually in Figure F.8(b). The switch fabric is described in greater detail in Sections F.4 through F.7, where various black box layers for switched-media networks are further revealed. Nevertheless, the high-level view shown

in Figure F.8(b) illustrates the potential bandwidth improvement of switched-media networks over shared-media networks: aggregate bandwidth can be many times higher than that of shared-media networks, allowing the possibility of greater effective bandwidth to be achieved. At best, only one node at a time can transmit packets over the shared media, whereas it is possible for all attached nodes to do so over the switched-media network.

Like their shared-media counterparts, switched-media networks must implement the three additional functions previously mentioned: routing, arbitration, and switching. Every time a packet enters the network, it is routed in order to select a path toward its destination provided by the topology. The path requested by the packet must be granted by some centralized or distributed arbiter, which resolves conflicts among concurrent requests for resources along the same path. Once the requested resources are granted, the network “switches in” the required connections to establish the path and allows the packet to be forwarded toward its destination. If the requested resources are not granted, the packet is usually buffered, as mentioned previously. Routing, arbitration, and switching functions are usually performed within switched networks in this order, whereas in shared-media networks routing typically is the last function performed.

Comparison of Shared- and Switched-Media Networks

In general, the advantage of shared-media networks is their low cost, but, consequently, their aggregate network bandwidth does not scale at all with the number of interconnected devices. Also, a global arbitration scheme is required to resolve conflicting demands, possibly introducing another type of bottleneck and again limiting scalability. Moreover, every device attached to the shared media increases the parasitic capacitance of the electrical conductors, thus increasing the time of flight propagation delay accordingly and, possibly, clock cycle time. In addition, it is more difficult to pipeline packet transmission over the network as the shared media are continuously granted to different requesting devices.

The main advantage of switched-media networks is that the amount of network resources implemented scales with the number of connected devices, increasing the aggregate network bandwidth. These networks allow multiple pairs of nodes to communicate simultaneously, allowing much higher effective network bandwidth than that provided by shared-media networks. Also, switched-media networks allow the system to scale to very large numbers of nodes, which is not feasible when using shared media. Consequently, this scaling advantage can, at the same time, be a disadvantage if network resources grow superlinearly. Networks of superlinear cost that provide an effective network bandwidth that grows only sublinearly with the number of interconnected devices are inefficient designs for many applications and interconnection network domains.

Characterizing Performance: Latency and Effective Bandwidth

The routing, switching, and arbitration functionality described above introduces some additional components of packet transport latency that must be taken into

account in the expression for total packet latency. Assuming there is no contention for network resources—as would be the case in an unloaded network—total packet latency is given by the following:

$$\text{Latency} = \text{Sending overhead} + (T_{\text{TotalProp}} + T_R + T_A + T_S) + \frac{\text{Packet size}}{\text{Bandwidth}} + \text{Receiving overhead}$$

Here T_R , T_A , and T_S are the total routing time, arbitration time, and switching time experienced by the packet, respectively, and are either measured quantities or calculated quantities derived from more detailed analyses. These components are added to the total propagation delay through the network links, $T_{\text{TotalProp}}$, to give the overall time of flight of the packet.

The expression above gives only a lower bound for the total packet latency as it does not account for additional delays due to contention for resources that may occur. When the network is heavily loaded, several packets may request the same network resources concurrently, thus causing contention that degrades performance. Packets that lose arbitration have to be buffered, which increases packet latency by some *contention delay* amount of waiting time. This additional delay is not included in the above expression. When the network or part of it approaches saturation, contention delay may be several orders of magnitude greater than the total packet latency suffered by a packet under zero load or even under slightly loaded network conditions. Unfortunately, it is not easy to compute analytically the total packet latency when the network is more than moderately loaded. Measurement of these quantities using cycle-accurate simulation of a detailed network model is a better and more precise way of estimating packet latency under such circumstances. Nevertheless, the expression given above is useful in calculating *best-case lower bounds* for packet latency.

For similar reasons, effective bandwidth is not easy to compute exactly, but we can estimate *best-case upper bounds* for it by appropriately extending the model presented at the end of the previous section. What we need to do is to find the narrowest section of the end-to-end network pipe by finding the network injection bandwidth ($BW_{\text{NetworkInjection}}$), the network reception bandwidth ($BW_{\text{NetworkReception}}$), and the network bandwidth (BW_{Network}) across the entire network interconnecting the devices.

The $BW_{\text{NetworkInjection}}$ can be calculated simply by multiplying the expression for link injection bandwidth, $BW_{\text{LinkInjection}}$, by the total number of network injection links. The $BW_{\text{NetworkReception}}$ is calculated similarly using $BW_{\text{LinkReception}}$, but it must also be scaled by a factor that reflects application traffic and other characteristics. For more than two interconnected devices, it is no longer valid to assume a one-to-one relationship among sources and destinations when analyzing the effect of flow control on link reception bandwidth. It could happen, for example, that several packets from different injection links arrive concurrently at the same reception link for applications that have many-to-one traffic characteristics, which causes contention at the reception links. This effect can be taken into account by an *average reception factor* parameter, σ , which is either a measured quantity or a calculated quantity derived from detailed analysis. It is defined as the average

fraction or percentage of packets arriving at reception links that can be accepted. Only those packets can be immediately delivered, thus reducing network reception bandwidth by that factor. This reduction occurs as a result of application behavior regardless of internal network characteristics. Finally, $BW_{Network}$ takes into account the internal characteristics of the network, including contention. We will progressively derive expressions in the following sections that will enable us to calculate this as more details are revealed about the internals of our black box interconnection network.

Overall, the effective bandwidth delivered by the network end-to-end to an application is determined by the minimum across the three sections, as described by the following:

$$\begin{aligned}\text{Effective bandwidth} &= \min(BW_{NetworkInjection}, BW_{Network}, \sigma \times BW_{NetworkReception}) \\ &= \min(N \times BW_{LinkInjection}, BW_{Network}, \sigma \times N \times BW_{LinkReception})\end{aligned}$$

Let's use the above expressions to compare the latency and effective bandwidth of shared-media networks against switched-media networks for the four interconnection network domains: OCNs, SANs, LANs, and WANs.

Example Plot the total packet latency and effective bandwidth as the number of interconnected nodes, N , scales from 4 to 1024 for shared-media and switched-media OCNs, SANs, LANs, and WANs. Assume that all network links, including the injection and reception links at the nodes, each have a data bandwidth of 8 Gbps, and unicast packets of 100 bytes are transmitted. Shared-media networks share one link, and switched-media networks have at least as many network links as there are nodes. For both, ignore latency and bandwidth effects due to contention within the network. End nodes have per-packet sending and receiving overheads of $x + 0.05$ ns/byte and $4/3(x) + 0.05$ ns/byte, respectively, where x is 0 μ s for the OCN, 0.3 μ s for the SAN, 3 μ s for the LAN, and 30 μ s for the WAN, and interconnection distances are 0.5 cm, 5 m, 5000 m, and 5000 km, respectively. Also assume that the total routing, arbitration, and switching times are constants or functions of the number of interconnected nodes: $T_R = 2.5$ ns, $T_A = 2.5(N)$ ns, and $T_S = 2.5$ ns for shared-media networks and $T_R = T_A = T_S = 2.5(\log_2 N)$ ns for switched-media networks. Finally, taking into account application traffic characteristics for the network structure, the average reception factor, σ , is assumed to be N^{-1} for shared media and polylogarithmic $(\log_2 N)^{-1/4}$ for switched media.

Answer All components of total packet latency are the same as in the example given in the previous section except for time of flight, which now has additional routing, arbitration, and switching delays. For shared-media networks, the additional delays total $5 + 2.5(N)$ ns; for switched-media networks, they total $7.5(\log_2 N)$ ns. Latency is plotted only for OCNs and SANs in Figure F.9 as these networks give the more interesting results. For OCNs, T_R , T_A , and T_S combine to dominate time of flight

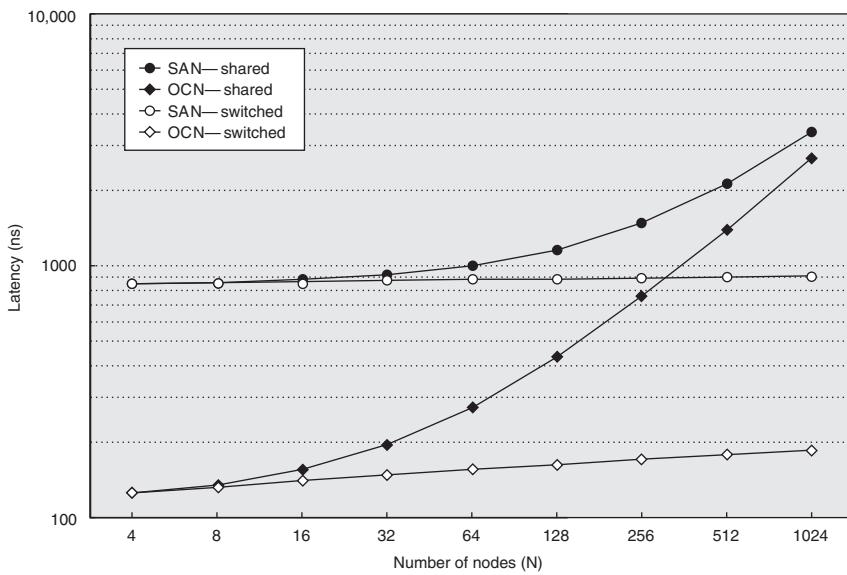


Figure F.9 Latency versus number of interconnected nodes plotted in semi-log form for OCNs and SANs. Routing, arbitration, and switching have more of an impact on latency for networks in these two domains, particularly for networks with a large number of nodes, given the low sending and receiving overheads and low propagation delay.

and are much greater than each of the other latency components for a moderate to large number of nodes. This is particularly so for the shared-media network. The latency increases much more dramatically with the number of nodes for shared media as compared to switched media given the difference in arbitration delay between the two. For SANs, T_R , T_A , and T_S dominate time of flight for most network sizes but are greater than each of the other latency components in shared-media networks only for large-sized networks; they are less than the other latency components for switched-media networks but are not negligible. For LANs and WANs, time of flight is dominated by propagation delay, which dominates other latency components as calculated in the previous section; thus, T_R , T_A , and T_S are negligible for both shared and switched media.

Figure F.10 plots effective bandwidth versus number of interconnected nodes for the four network domains. The effective bandwidth for all shared-media networks is constant through network scaling as only one unicast packet can be received at a time over all the network reception links, and that is further limited by the receiving overhead of each network for all but the OCN. The effective bandwidth for all switched-media networks increases with the number of interconnected nodes, but it is scaled down by the average reception factor. The receiving overhead further limits effective bandwidth for all but the OCN.

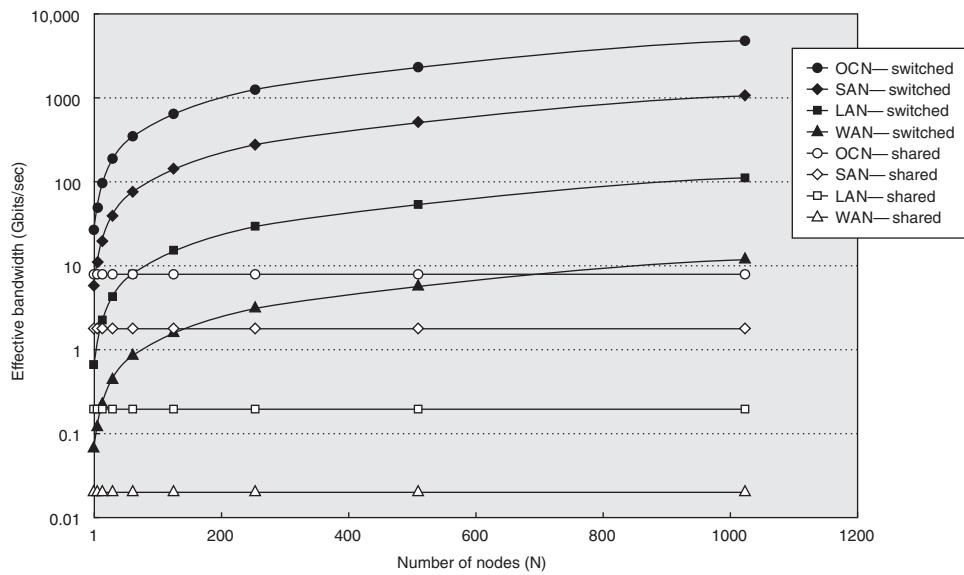


Figure F.10 Effective bandwidth versus number of interconnected nodes plotted in semi-log form for the four network domains. The disparity in effective bandwidth between shared- and switched-media networks for all interconnect domains widens significantly as the number of nodes in the network increases. Only the switched on-chip network is able to achieve an effective bandwidth equal to the aggregate bandwidth for the parameters given in this example.

Given the obvious advantages, why weren't switched networks always used? Earlier computers were much slower and could share the network media with little impact on performance. In addition, the switches for earlier LANs and WANs took up several large boards and were about as large as an entire computer. As a consequence of Moore's law, the size of switches has reduced considerably, and systems have a much greater need for high-performance communication. Switched networks allow communication to harvest the same rapid advancements from silicon as processors and main memory. Whereas switches from telecommunication companies were once the size of mainframe computers, today we see single-chip switches and even entire switched networks within a chip. Thus, technology and application trends favor switched networks today. Just as single-chip processors led to processors replacing logic circuits in a surprising number of places, single-chip switches and switched on-chip networks are increasingly replacing shared-media networks (i.e., buses) in several application domains. As an example, PCI-Express (PCIe)—a switched network—was introduced in 2005 to replace the traditional PCI-X bus on personal computer motherboards.

The previous example also highlights the importance of optimizing the routing, arbitration, and switching functions in OCNs and SANs. For these network domains in particular, the interconnect distances and overheads typically are small

enough to make latency and effective bandwidth much more sensitive to how well these functions are implemented, particularly for larger-sized networks. This leads mostly to implementations based mainly on the faster hardware solutions for these domains. In LANs and WANs, implementations based on the slower but more flexible software solutions suffice given that performance is largely determined by other factors. The design of the topology for switched-media networks also plays a major role in determining how close to the lower bound on latency and the upper bound on effective bandwidth the network can achieve for OCN and SAN domains.

The next three sections touch on these important issues in switched networks, with the next section focused on topology.

F.4

Network Topology

When the number of devices is small enough, a single switch is sufficient to interconnect them within a switched-media network. However, the number of *switch ports* is limited by existing very-large-scale integration (VLSI) technology, cost considerations, power consumption, and so on. When the number of required *network ports* exceeds the number of ports supported by a single switch, a fabric of interconnected switches is needed. To embody the necessary property of *full access* (i.e., *connectedness*), the network switch fabric must provide a path from every end node device to every other device. All the connections to the network fabric and between switches within the fabric use point-to-point links as opposed to shared links—that is, links with only one switch or end node device on either end. The interconnection structure across all the components—including switches, links, and end node devices—is referred to as the *network topology*.

The number of network topologies described in the literature would be difficult to count, but the number that have been used commercially is no more than about a dozen or so. During the 1970s and early 1980s, researchers struggled to propose new topologies that could reduce the number of switches through which packets must traverse, referred to as the *hop count*. In the 1990s, thanks to the introduction of pipelined transmission and switching techniques, the hop count became less critical. Nevertheless, today, topology is still important, particularly for OCNs and SANs, as subtle relationships exist between topology and other network design parameters that impact performance, especially when the number of end nodes is very large (e.g., 64 K in the Blue Gene/L supercomputer) or when the latency is critical (e.g., in multicore processor chips). Topology also greatly impacts the implementation cost of the network.

Topologies for parallel supercomputer SANs have been the most visible and imaginative, usually converging on regularly structured ones to simplify routing, packaging, and scalability. Those for LANs and WANs tend to be more haphazard or ad hoc, having more to do with the challenges of long distance or connecting across different communication subnets. Switch-based topologies for OCNs are only recently emerging but are quickly gaining in popularity. This section

describes the more popular topologies used in commercial products. Their advantages, disadvantages, and constraints are also briefly discussed.

Centralized Switched Networks

As mentioned above, a single switch suffices to interconnect a set of devices when the number of switch ports is equal to or larger than the number of devices. This simple network is usually referred to as a *crossbar* or *crossbar switch*. Within the crossbar, crosspoint switch complexity increases quadratically with the number of ports, as illustrated in Figure F.11(a). Thus, a cheaper solution is desirable when the number of devices to be interconnected scales beyond the point supportable by implementation technology.

A common way of addressing the crossbar scaling problem consists of splitting the large crossbar switch into several stages of smaller switches interconnected in such a way that a single pass through the switch fabric allows any destination to be reached from any source. Topologies arranged in this way are usually referred to as *multistage interconnection networks* or *multistage switch fabrics*, and these networks typically have complexity that increases in proportion to $N \log N$. Multistage interconnection networks (MINs) were initially proposed for telephone exchanges in the 1950s and have since been used to build the communication backbone for parallel supercomputers, symmetric multiprocessors, multicenter clusters, and IP router switch fabrics.

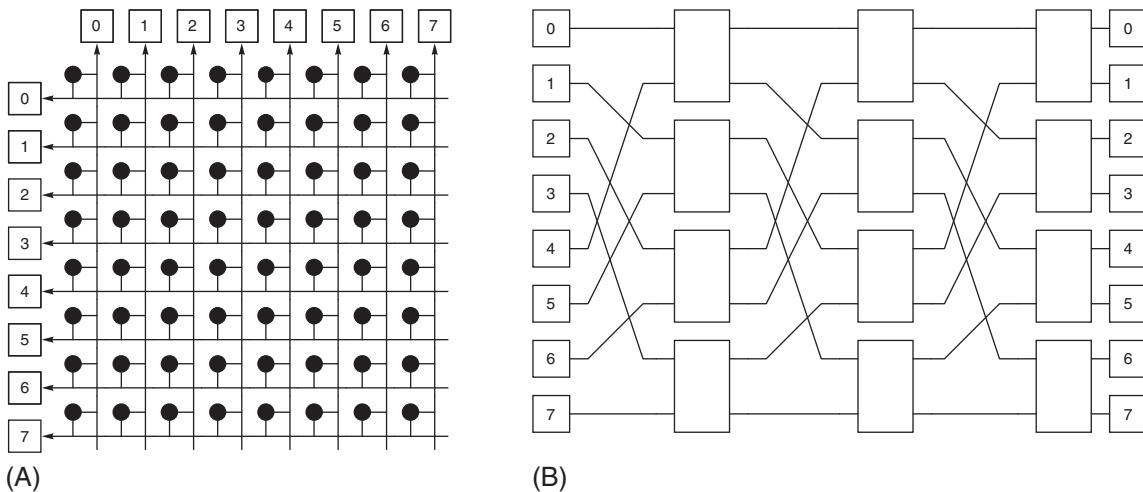


Figure F.11 Popular centralized switched networks: (a) the crossbar network requires N^2 crosspoint switches, shown as black dots; (b) the Omega, a MIN, requires $N/2 \log_2 N$ switches, shown as vertical rectangles. End node devices are shown as numbered squares (total of eight). Links are unidirectional—data enter at the left and exit out the top or right.

The interconnection pattern or patterns between MIN stages are permutations that can be represented mathematically by a set of functions, one for each stage. Figure F.11(b) shows a well-known MIN topology, the *Omega*, which uses the perfect-shuffle permutation as its interconnection pattern for each stage, followed by exchange switches, giving rise to a *perfect-shuffle exchange* for each stage. In this example, eight input-output ports are interconnected with three stages of 2×2 switches. It is easy to see that a single pass through the three stages allows any input port to reach any output port. In general, when using $k \times k$ switches, a MIN with N input-output ports requires at least $\log_k N$ stages, each of which contains N/k switches, for a total of N/k ($\log_k N$) switches.

Despite their internal structure, MINs can be seen as centralized switch fabrics that have end node devices connected at the network periphery, hence the name *centralized switched network*. From another perspective, MINs can be viewed as interconnecting nodes through a set of switches that may not have any nodes directly connected to them, which gives rise to another popular name for centralized switched networks—*indirect networks*.

Example Compute the cost of interconnecting 4096 nodes using a single crossbar switch relative to doing so using a MIN built from 2×2 , 4×4 , and 16×16 switches. Consider separately the relative cost of the unidirectional links and the relative cost of the switches. Switch cost is assumed to grow quadratically with the number of input (alternatively, output) ports, k , for $k \times k$ switches.

Answer The switch cost of the network when using a single crossbar is proportional to 4096^2 . The unidirectional link cost is 8192, which accounts for the set of links from the end nodes to the crossbar and also from the crossbar back to the end nodes. When using a MIN with $k \times k$ switches, the cost of each switch is proportional to k^2 but there are $4096/k$ ($\log_k 4096$) total switches. Likewise, there are ($\log_k 4096$) stages of N unidirectional links per stage from the switches plus N links to the MIN from the end nodes. Therefore, the relative costs of the crossbar with respect to each MIN is given by the following:

$$\text{Relative cost } (2 \times 2)_{\text{switches}} = 4096^2 / (2^2 \times 4096/2 \times \log_2 4096) = 170$$

$$\text{Relative cost } (4 \times 4)_{\text{switches}} = 4096^2 / (4^2 \times 4096/4 \times \log_4 4096) = 170$$

$$\text{Relative cost } (16 \times 16)_{\text{switches}} = 4096^2 / (16^2 \times 4096/16 \times \log_{16} 4096) = 85$$

$$\text{Relative cost } (2 \times 2)_{\text{links}} = 8192 / (4096 \times (\log_2 4096 + 1)) = 2/13 = 0.1538$$

$$\text{Relative cost } (4 \times 4)_{\text{links}} = 8192 / (4096 \times (\log_4 4096 + 1)) = 2/7 = 0.2857$$

$$\text{Relative cost } (16 \times 16)_{\text{links}} = 8192 / (4096 \times (\log_{16} 4096 + 1)) = 2/4 = 0.5$$

In all cases, the single crossbar has much higher switch cost than the MINs. The most dramatic reduction in cost comes from the MIN composed from the smallest sized but largest number of switches, but it is interesting to see that the MINs with 2×2 and 4×4 switches yield the same relative switch cost. The relative link cost

of the crossbar is lower than the MINs, but by less than an order of magnitude in all cases. We must keep in mind that end node links are different from switch links in their length and packaging requirements, so they usually have different associated costs. Despite the lower link cost, the crossbar has higher overall relative cost.

The reduction in switch cost of MINs comes at the price of performance: contention is more likely to occur on network links, thus degrading performance. Contention in the form of packets *blocking* in the network arises due to paths from different sources to different destinations simultaneously sharing one or more links. The amount of contention in the network depends on communication traffic behavior. In the Omega network shown in Figure F.11(b), for example, a packet from port 0 to port 1 blocks in the first stage of switches while waiting for a packet from port 4 to port 0. In the crossbar, no such blocking occurs as links are not shared among paths to unique destinations. The crossbar, therefore, is *nonblocking*. Of course, if two nodes try to send packets to the same destination, there will be blocking at the reception link even for crossbar networks. This is accounted for by the average reception factor parameter (σ) when analyzing performance, as discussed at the end of the previous section.

To reduce blocking in MINs, extra switches must be added or larger ones need to be used to provide alternative paths from every source to every destination. The first commonly used solution is to add a minimum of $\log_k N - 1$ extra switch stages to the MIN in such a way that they mirror the original topology. The resulting network is *rearrangeably nonblocking* as it allows nonconflicting paths among new source-destination pairs to be established, but it also doubles the hop count and could require the paths of some existing communicating pairs to be rearranged under some centralized control. The second solution takes a different approach. Instead of using more switch stages, larger switches—which can be implemented by multiple stages if desired—are used in the middle of two other switch stages in such a way that enough alternative paths through the middle-stage switches allow for nonconflicting paths to be established between the first and last stages. The best-known example of this is the Clos network, which is nonblocking. The multipath property of the three-stage Clos topology can be recursively applied to the middle-stage switches to reduce the size of all the switches down to 2×2 , assuming that switches of this size are used in the first and last stages to begin with. What results is a Beneš topology consisting of $2(\log_2 N) - 1$ stages, which is rearrangeably nonblocking. Figure F.12(a) illustrates both topologies, where all switches not in the first and last stages comprise the middle-stage switches (recursively) of the Clos network.

The MINs described so far have unidirectional network links, but bidirectional forms are easily derived from symmetric networks such as the Clos and Beneš simply by folding them. The overlapping unidirectional links run in different directions, thus forming bidirectional links, and the overlapping switches merge into a single switch with twice the ports (i.e., 4×4 switch). Figure F.12(b) shows the resulting folded Beneš topology but in this case with the end nodes connected

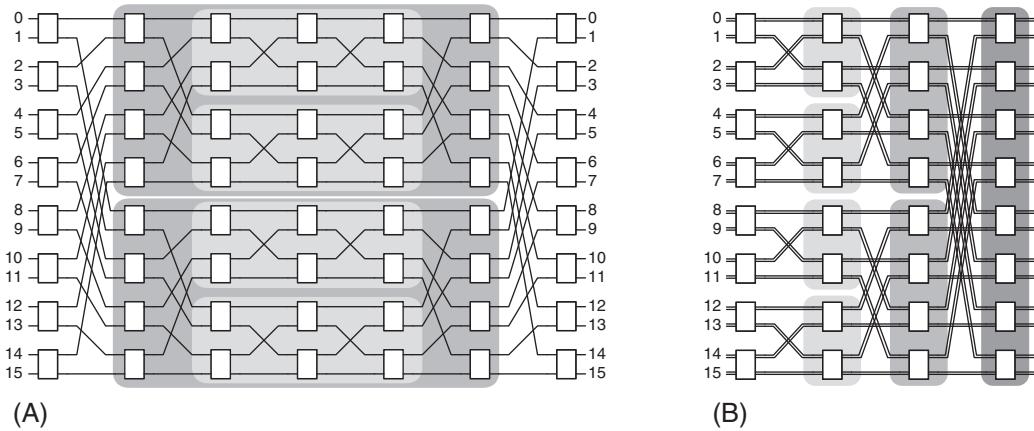


Figure F.12 Two Beneš networks. (a) A 16-port Clos topology, where the middle-stage switches shown in the darker shading are implemented with another Clos network whose middle-stage switches shown in the lighter shading are implemented with yet another Clos network, and so on, until a Beneš network is produced that uses only 2×2 switches everywhere. (b) A folded Beneš network (bidirectional) in which 4×4 switches are used; end nodes attach to the innermost set of the Beneš network (unidirectional) switches. This topology is equivalent to a fat tree, where tree vertices are shown in shades.

to the innermost switch stage of the original Beneš. Ports remain free at the other side of the network but can be used for later expansion of the network to larger sizes. These kind of networks are referred to as *bidirectional multistage interconnection networks*. Among many useful properties of these networks are their modularity and their ability to exploit communication locality, which saves packets from having to hop across all network stages. Their regularity also reduces routing complexity and their multipath property enables traffic to be routed more evenly across network resources and to tolerate faults.

Another way of deriving bidirectional MINs with nonblocking (rearrangeable) properties is to form a balanced tree, where end node devices occupy leaves of the tree and switches occupy vertices within the tree. Enough links in each tree level must be provided such that the total link bandwidth remains constant across all levels. Also, except for the root, switch ports for each vertex typically grow as $k^i \times k^i$, where i is the tree level. This can be accomplished by using k^{i-1} total switches at each vertex, where each switch has k input and k output ports, or k bidirectional ports (i.e., $k \times k$ input-output ports). Networks having such topologies are called *fat tree* networks. As only half of the k bidirectional ports are used in each direction, $2 N/k$ switches are needed in each stage, totaling $2 N/k (\log_{k/2} N)$ switches in the fat tree. The number of switches in the root stage can be halved as no forward links are needed, reducing switch count by N/k . Figure F.12(b) shows a fat tree for 4×4 switches. As can be seen, this is identical to the folded Beneš.

The fat tree is the topology of choice across a wide range of network sizes for most commercial systems that use multistage interconnection networks. Most SANs used in multicomputer clusters, and many used in the most powerful

supercomputers, are based on fat trees. Commercial communication subsystems offered by Myrinet, Mellanox, and Quadrics are also built from fat trees.

Distributed Switched Networks

Switched-media networks provide a very flexible framework to design communication subsystems external to the devices that need to communicate, as presented above. However, there are cases where it is convenient to more tightly integrate the end node devices with the network resources used to enable them to communicate. Instead of centralizing the switch fabric in an external subsystem, an alternative approach is to distribute the network switches among the end nodes, which then become *network nodes* or simply *nodes*, yielding a *distributed switched network*. As a consequence, each network switch has one or more end node devices directly connected to it, thus forming a network node. These nodes are directly connected to other nodes without indirectly going through some external switch, giving rise to another popular name for these networks—*direct networks*.

The topology for distributed switched networks takes on a form much different from centralized switched networks in that end nodes are connected across the area of the switch fabric, not just at one or two of the peripheral edges of the fabric. This causes the number of switches in the system to be equal to the total number of nodes. A quite obvious way of interconnecting nodes consists of connecting a dedicated link between each node and every other node in the network. This *fully connected* topology provides the best connectivity (full connectivity in fact), but it is more costly than a crossbar network, as the following example shows.

Example Compute the cost of interconnecting N nodes using a fully connected topology relative to doing so using a crossbar topology. Consider separately the relative cost of the unidirectional links and the relative cost of the switches. Switch cost is assumed to grow quadratically with the number of unidirectional ports for $k \times k$ switches but to grow only linearly with $1 \times k$ switches.

Answer The crossbar topology requires an $N \times N$ switch, so the switch cost is proportional to N^2 . The link cost is $2N$, which accounts for the unidirectional links from the end nodes to the centralized crossbar, and *vice versa*. In the fully connected topology, two sets of $1 \times (N - 1)$ switches (possibly merged into one set) are used in each of the N nodes to connect nodes directly to and from all other nodes. Thus, the total switch cost for all N nodes is proportional to $2N(N - 1)$. Regarding link cost, each of the N nodes requires two unidirectional links in opposite directions between its end node device and its local switch. In addition, each of the N nodes has $N - 1$ unidirectional links from its local switch to other switches distributed across all the other end nodes. Thus, the total number of unidirectional links is $2N + N(N - 1)$, which is equal to $N(N + 1)$ for all N nodes. The relative costs of the fully connected topology with respect to the crossbar is, therefore, the following:

$$\text{Relative cost}_{\text{switches}} = 2N(N-1)/N^2 = 2(N-1)/N = 2(1 - 1/N)$$

$$\text{Relative cost}_{\text{links}} = N(N+1)/2N = (N+1)/2$$

As the number of interconnected devices increases, the switch cost of the fully connected topology is nearly double the crossbar, with both being very high (i.e., quadratic growth). Moreover, the fully connected topology always has higher relative link cost, which grows linearly with the number of nodes. Again, keep in mind that end node links are different from switch links in their length and packaging, particularly for direct networks, so they usually have different associated costs. Despite its higher cost, the fully connected topology provides no extra performance benefits over the crossbar as both are nonblocking. Thus, crossbar networks are usually used in practice instead of fully connected networks.

A lower-cost alternative to fully connecting all nodes in the network is to directly connect nodes in sequence along a *ring* topology, as shown in Figure F.13. For bidirectional rings, each of the N nodes now uses only 3×3 switches and just two bidirectional network links (shared by neighboring nodes), for a total of N switches and N bidirectional network links. This linear cost excludes the N injection-reception bidirectional links required within nodes.

Unlike shared-media networks, rings can allow many simultaneous transfers: the first node can send to the second while the second sends to the third, and so on. However, as dedicated links do not exist between logically nonadjacent node pairs, packets must hop across intermediate nodes before arriving at their destination, increasing their transport latency. For bidirectional rings, packets can be transported in either direction, with the shortest path to the destination usually being the one selected. In this case, packets must travel $N/4$ network switch hops, on average, with total switch hop count being one more to account for the local switch at the packet source node. Along the way, packets may block on network resources due to other packets contending for the same resources simultaneously.

Fully connected and ring-connected networks delimit the two extremes of distributed switched topologies, but there are many points of interest in between for a given set of cost-performance requirements. Generally speaking, the ideal switched-media topology has cost approaching that of a ring but performance

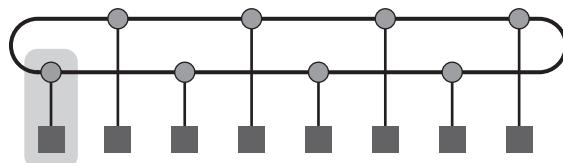


Figure F.13 A ring network topology, folded to reduce the length of the longest link. Shaded circles represent switches, and black squares represent end node devices. The gray rectangle signifies a network node consisting of a switch, a device, and its connecting link.

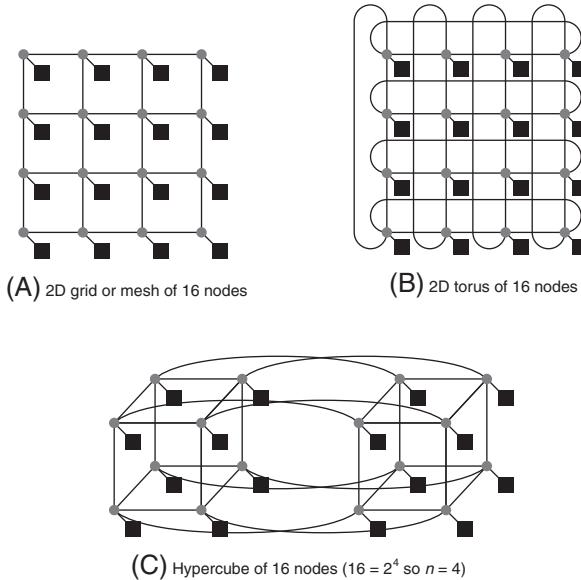


Figure F.14 Direct network topologies that have appeared in commercial systems, mostly supercomputers.

The shaded circles represent switches, and the black squares represent end node devices. Switches have many bidirectional network links, but at least one link goes to the end node device. These basic topologies can be supplemented with extra links to improve performance and reliability. For example, connecting the switches on the periphery of the 2D mesh, shown in (a), using the unused ports on each switch forms a 2D torus, shown in (b). The hypercube topology, shown in (c) is an n -dimensional interconnect for 2^n nodes, requiring $n+1$ ports per switch: one for the n nearest neighbor nodes and one for the end node device.

approaching that of a fully connected topology. Figure F.14 illustrates three popular direct network topologies commonly used in systems spanning the cost-performance spectrum. All of them consist of sets of nodes arranged along multiple dimensions with a regular interconnection pattern among nodes that can be expressed mathematically. In the *mesh* or *grid* topology, all the nodes in each dimension form a linear array. In the *torus* topology, all the nodes in each dimension form a ring. Both of these topologies provide direct communication to neighboring nodes with the aim of reducing the number of hops suffered by packets in the network with respect to the ring. This is achieved by providing greater connectivity through additional dimensions, typically no more than three in commercial systems. The *hypercube* or *n-cube* topology is a particular case of the mesh in which only two nodes are interconnected along each dimension, leading to a number of dimensions, n , that must be large enough to interconnect all N nodes in the system (i.e., $n = \log_2 N$). The hypercube provides better connectivity than meshes

and tori at the expense of higher link and switch costs, in terms of the number of links and number of ports per node.

Example Compute the cost of interconnecting N devices using a torus topology relative to doing so using a fat tree topology. Consider separately the relative cost of the bidirectional links and the relative cost of the switches—which is assumed to grow quadratically with the number of bidirectional ports. Provide an approximate expression for the case of switches being similar in size.

Answer Using $k \times k$ switches, the fat tree requires $2N/k (\log_{k/2} N)$ switches, assuming the last stage (the root) has the same number of switches as each of the other stages. Given that the number of bidirectional ports in each switch is k (i.e., there are k input ports and k output ports for a $k \times k$ switch) and that the switch cost grows quadratically with this, total network switch cost is proportional to $2kN \log_{k/2} N$. The link cost is $N \log_{k/2} N$ as each of the $\log_{k/2} N$ stages requires N bidirectional links, including those between the devices and the fat tree. The torus requires as many switches as nodes, each of them having $2n+1$ bidirectional ports, including the port to attach the communicating device, where n is the number of dimensions. Hence, total switch cost for the torus is $(2n+1)^2 N$. Each of the torus nodes requires $2n+1$ bidirectional links for the n different dimensions and the connection for its end node device, but as the dimensional links are shared by two nodes, the total number of links is $(2n/2+1)N = (n+1)N$ bidirectional links for all N nodes. Thus, the relative costs of the torus topology with respect to the fat tree are

$$\text{Relative cost}_{\text{switches}} = (2n+1)^2 N / 2kN \log_{k/2} N = (2n+1)^2 / 2k \log_{k/2} N$$

$$\text{Relative cost}_{\text{links}} = (n+1)N / N \log_{k/2} N = (n+1) / \log_{k/2} N$$

When switch sizes are similar, $2n+1 \approx k$. In this case, the relative cost is

$$\text{Relative cost}_{\text{switches}} = (2n+1)^2 / 2k \log_{k/2} N = (2n+1) / 2 \log_{k/2} N = k / 2 \log_{k/2} N$$

When the number of switch ports (also called *switch degree*) is small, tori have lower cost, particularly when the number of dimensions is low. This is an especially useful property when N is large. On the other hand, when larger switches and/or a high number of tori dimensions are used, fat trees are less costly and preferable. For example, when interconnecting 256 nodes, a fat tree is four times more expensive in terms of switch and link costs when 4×4 switches are used. This higher cost is compensated for by lower network contention, on average. The fat tree is comparable in cost to the torus when 8×8 switches are used (e.g., for interconnecting 256 nodes). For larger switch sizes beyond this, the torus costs more than the fat tree as each node includes a switch. This cost can be amortized by connecting multiple end node devices per switch, called *bristling*.

The topologies depicted in Figure F.14 all have in common the interesting characteristic of having their network links arranged in several orthogonal dimensions in a regular way. In fact, these topologies all happen to be particular

instances of a larger class of direct network topologies known as k -ary n -cubes, where k signifies the number of nodes interconnected in each of the n dimensions. The symmetry and regularity of these topologies simplify network implementation (i.e., packaging) and packet routing as the movement of a packet along a given network dimension does not modify the number of remaining hops in any other dimension toward its destination. As we will see in the next section, this topological property can be readily exploited by simple routing algorithms.

Like their indirect counterpart, direct networks can introduce blocking among packets that concurrently request the same path, or part of it. The only exception is fully connected networks. The same way that the number of stages and switch hops in indirect networks can be reduced by using larger switches, the hop count in direct networks can likewise be reduced by increasing the number of topological dimensions via increased switch degree.

It may seem to be a good idea always to maximize the number of dimensions for a system of a certain size and switch cost. However, this is not necessarily the case. Most electronic systems are built within our three-dimensional (3D) world using planar (2D) packaging technology such as integrated circuit chips, printed circuit boards, and backplanes. Direct networks with up to three dimensions can be implemented using relatively short links within this 3D space, independent of system size. Links in higher-dimensioned networks would require increasingly longer wires or fiber. This increase in link length with system size is also indicative of MINs, including fat trees, which require either long links within all the stages or increasingly longer links as more stages are added. As we saw in the first example given in Section F.2, flow-controlled buffers increase in size proportionally to link length, thus requiring greater silicon area. This is among the reasons why the supercomputer with the largest number of compute nodes existing in 2005, the IBM Blue Gene/L, implemented a 3D torus network for interprocessor communication. A fat tree would have required much longer links, rendering a 64K node system less feasible. This highlights the importance of correctly selecting the proper network topology that meets system requirements.

Besides link length, other constraints derived from implementing the topology may also limit the degree to which a topology can scale. These are available *pin-out* and achievable *bisection bandwidth*. Pin count is a local restriction on the bandwidth of a chip, printed circuit board, and backplane (or chassis) connector. In a direct network that integrates processor cores and switches on a single chip or multichip module, pin bandwidth is used both for interfacing with main memory and for implementing node links. In this case, limited pin count could reduce the number of switch ports or bit lines per link. In an indirect network, switches are implemented separately from processor cores, allowing most of the pins to be dedicated to communication bandwidth. However, as switches are grouped onto boards, the aggregate of all input-output links of the switch fabric on a board for a given topology must not exceed the board connector pin-outs.

The bisection bandwidth is a more global restriction that gives the interconnect density and bandwidth that can be achieved by a given implementation

(packaging) technology. Interconnect density and clock frequency are related to each other: When wires are packed closer together, crosstalk and parasitic capacitance increase, which usually impose a lower clock frequency. For example, the availability and spacing of metal layers limit wire density and frequency of on-chip networks, and copper track density limits wire density and frequency on a printed circuit board. To be implementable, the topology of a network must not exceed the available bisection bandwidth of the implementation technology. Most networks implemented to date are constrained more so by pin-out limitations rather than bisection bandwidth, particularly with the recent move to blade-based systems. Nevertheless, bisection bandwidth largely affects performance.

For a given topology, bisection bandwidth, $BW_{\text{Bisection}}$, is calculated by dividing the network into two roughly equal parts—each with half the nodes—and summing the bandwidth of the links crossing the imaginary dividing line. For nonsymmetric topologies, bisection bandwidth is the smallest of all pairs of equal-sized divisions of the network. For a fully connected network, the bisection bandwidth is proportional to $N^2/2$ unidirectional links (or $N^2/4$ bidirectional links), where N is the number of nodes. For a bus, bisection bandwidth is the bandwidth of just the one shared half-duplex link. For other topologies, values lie in between these two extremes. Network injection and reception bisection bandwidth is commonly used as a reference value, which is $N/2$ for a network with N injection and reception links, respectively. Any network topology that provides this bisection bandwidth is said to have *full bisection bandwidth*.

Figure F.15 summarizes the number of switches and links required, the corresponding switch size, the maximum and average switch hop distances between nodes, and the bisection bandwidth in terms of links for several topologies discussed in this section for interconnecting 64 nodes.

Evaluation category	Bus	Ring	2D mesh	2D torus	Hypercube	Fat tree	Fully connected
Performance							
$BW_{\text{Bisection}}$ in # links	1	2	8	16	32	32	1024
Max (ave.) hop count	1 (1)	32 (16)	14 (7)	8 (4)	6 (3)	11 (9)	1 (1)
Cost							
I/O ports per switch	NA	3	5	5	7	4	64
Number of switches	NA	64	64	64	64	192	64
Number of net. links	1	64	112	128	192	320	2016
Total number of links	1	128	176	192	256	384	2080

Figure F.15 Performance and cost of several network topologies for 64 nodes. The bus is the standard reference at unit network link cost and bisection bandwidth. Values are given in terms of bidirectional links and ports. Hop count includes a switch and its output link, but not the injection link at end nodes. Except for the bus, values are given for the number of network links and total number of links, including injection/reception links between end node devices and the network.

Effects of Topology on Network Performance

Switched network topologies require packets to take one or more hops to reach their destination, where each hop represents the transport of a packet through a switch and one of its corresponding links. Interestingly, each switch and its corresponding links can be modeled as a black box network connecting more than two devices, as was described in the previous section, where the term “devices” here refers to end nodes or other switches. The only differences are that the sending and receiving overheads are null through the switches, and the routing, switching, and arbitration delays are not cumulative but, instead, are delays associated with each switch.

As a consequence of the above, if the average packet has to traverse d hops to its destination, then $T_R + T_A + T_S = (T_r + T_a + T_s) \times d$, where T_r , T_a , and T_s are the routing, arbitration, and switching delays, respectively, of a switch. With the assumption that pipelining over the network is staged on each hop at the packet level (this assumption will be challenged in the next section), the transmission delay is also increased by a factor of the number of hops. Finally, with the simplifying assumption that all injection links to the first switch or stage of switches and all links (including reception links) from the switches have approximately the same length and delay, the total propagation delay through the network $T_{\text{TotalProp}}$ is the propagation delay through a single link, T_{LinkProp} , multiplied by $d+1$, which is the hop count plus one to account for the injection link. Thus, the best-case lower-bound expression for average packet latency in the network (i.e., the latency in the absence of contention) is given by the following expression:

$$\text{Latency} = \text{Sending overhead} + T_{\text{LinkProp}} \times (d+1) + (T_r + T_a + T_s) \times d + \frac{\text{Packet size}}{\text{Bandwidth}} \times (d+1) + \text{Receiving overhead}$$

Again, the expression on page F-40 assumes that switches are able to pipeline packet transmission at the packet level.

Following the method presented previously, we can estimate the best-case upper bound for effective bandwidth by finding the narrowest section of the end-to-end network pipe. Focusing on the internal network portion of that pipe, network bandwidth is determined by the blocking properties of the topology. Non-blocking behavior can be achieved only by providing many alternative paths between every source-destination pair, leading to an aggregate network bandwidth that is many times higher than the aggregate network injection or reception bandwidth. This is quite costly. As this solution usually is prohibitively expensive, most networks have different degrees of blocking, which reduces the utilization of the aggregate bandwidth provided by the topology. This, too, is costly but not in terms of performance.

The amount of blocking in a network depends on its topology and the traffic distribution. Assuming the bisection bandwidth, $BW_{\text{Bisection}}$, of a topology is implementable (as typically is the case), it can be used as a constant measure of the maximum degree of blocking in a network. In the ideal case, the network always achieves full bisection bandwidth irrespective of the traffic behavior, thus

transferring the bottlenecking point to the injection or reception links. However, as packets destined to locations in the other half of the network necessarily must cross the bisection links, those links pose as potential bottleneck links—potentially reducing the network bandwidth to below full bisection bandwidth. Fortunately, not all of the traffic must cross the network bisection, allowing more of the aggregate network bandwidth provided by the topology to be utilized. Also, network topologies with a higher number of bisection links tend to have less blocking as more alternative paths are possible to reach destinations and, hence, a higher percentage of the aggregate network bandwidth can be utilized. If only a fraction of the traffic must cross the network bisection, as captured by a *bisection traffic fraction* parameter γ ($0 < \gamma \leq 1$), the network pipe at the bisection is, effectively, widened by the reciprocal of that fraction, assuming a traffic distribution that loads the bisection links at least as heavily, on average, as other network links. This defines the upper limit on achievable network bandwidth, BW_{Network} :

$$BW_{\text{Network}} = \frac{BW_{\text{Bisection}}}{\gamma}$$

Accordingly, the expression for effective bandwidth becomes the following when network topology is taken into consideration:

$$\text{Effective bandwidth} = \min \left(N \times BW_{\text{LinkInjection}}, \frac{BW_{\text{Bisection}}}{\gamma}, \sigma \times N \times BW_{\text{LinkReception}} \right)$$

It is important to note that γ depends heavily on the traffic patterns generated by applications. It is a measured quantity or calculated from detailed traffic analysis.

Example A common communication pattern in scientific programs is to have nearest neighbor elements of a two-dimensional array to communicate in a given direction. This pattern is sometimes called *NEWS communication*, standing for north, east, west, and south—the directions on a compass. Map an 8×8 array of elements one-to-one onto 64 end node devices interconnected in the following topologies: bus, ring, 2D mesh, 2D torus, hypercube, fully connected, and fat tree. How long does it take in the best case for each node to send one message to its northern neighbor and one to its eastern neighbor, assuming packets are allowed to use any minimal path provided by the topology? What is the corresponding effective bandwidth? Ignore elements that have no northern or eastern neighbors. To simplify the analysis, assume that all networks experience unit packet transport time for each network hop—that is, T_{LinkProp} , T_r , T_a , T_s , and packet transmission time for each hop sum to one. Also assume the delay through injection links is included in this unit time, and sending/receiving overhead is null.

Answer This communication pattern requires us to send $2 \times (64 - 8)$ or 112 total packets—that is, 56 packets in each of the two communication phases: northward and eastward. The number of hops suffered by packets depends on the topology. Communication between sources and destinations are one-to-one, so σ is 100%.

The injection and reception bandwidth cap the effective bandwidth to a maximum of 64 BW units (even though the communication pattern requires only 56 BW units). However, this maximum may get scaled down by the achievable network bandwidth, which is determined by the bisection bandwidth and the fraction of traffic crossing it, γ , both of which are topology dependent. Here are the various cases:

- *Bus*—The mapping of the 8×8 array elements to nodes makes no difference for the bus as all nodes are equally distant at one hop away. However, the 112 transfers are done sequentially, taking a total of 112 time units. The bisection bandwidth is 1, and γ is 100%. Thus, effective bandwidth is only 1 BW unit.
- *Ring*—Assume the first row of the array is mapped to nodes 0 to 7, the second row to nodes 8 to 15, and so on. It takes just one time unit for all nodes simultaneously to send to their eastern neighbor (i.e., a transfer from node i to node $i+1$). With this mapping, the northern neighbor for each node is exactly eight hops away so it takes eight time units, which also is done in parallel for all nodes. Total communication time is, therefore, 9 time units. The bisection bandwidth is 2 bidirectional links (assuming a bidirectional ring), which is less than the full bisection bandwidth of 32 bidirectional links. For eastward communication, because only 2 of the eastward 56 packets must cross the bisection in the worst case, the bisection links do not pose as bottlenecks. For northward communication, 8 of the 56 packets must cross the two bisection links, yielding a γ of $8/112 = 7.14\%$. Thus, the network bandwidth is $2/0.0714 = 28.4$ BW units. This limits the effective bandwidth at 28.4 BW units as well, which is less than half the bandwidth required by the communication pattern.
- *2D mesh*—There are eight rows and eight columns in our grid of 64 nodes, which is a perfect match to the NEWS communication. It takes a total of just 2 time units for all nodes to send simultaneously to their northern neighbors followed by simultaneous communication to their eastern neighbors. The bisection bandwidth is 8 bidirectional links, which is less than full bisection bandwidth. However, the perfect matching of this nearest neighbor communication pattern on this topology allows the maximum effective bandwidth to be achieved regardless. For eastward communication, 8 of the 56 packets must cross the bisection in the worst case, which does not exceed the bisection bandwidth. None of the northward communications crosses the same network bisection, yielding a γ of $8/112 = 7.14\%$ and a network bandwidth of $8/0.0714 = 112$ BW units. The effective bandwidth is, therefore, limited by the communication pattern at 56 BW units as opposed to the mesh network.
- *2D torus*—Wrap-around links of the torus are not used for this communication pattern, so the torus has the same mapping and performance as the mesh.

- *Hypercube*—Assume elements in each row are mapped to the same location within the eight 3-cubes comprising the hypercube such that consecutive row elements are mapped to nodes only one hop away. Northern neighbors can be similarly mapped to nodes only one hop away in an orthogonal dimension. Thus, the communication pattern takes just 2 time units. The hypercube provides full bisection bandwidth of 32 links, but at most only 8 of the 112 packets must cross the bisection. Thus, effective bandwidth is limited only by the communication pattern to be 56 BW units, not by the hypercube network.
- *Fully connected*—Here, nodes are equally distant at one hop away, regardless of the mapping. Parallel transfer of packets in both the northern and eastern directions would take only 1 time unit if the injection and reception links could source and sink two packets at a time. As this is not the case, 2 time units are required. Effective bandwidth is limited by the communication pattern at 56 BW units, so the 1024 network bisection links largely go underutilized.
- *Fat tree*—Assume the same mapping of elements to nodes as is done for the ring and the use of switches with eight bidirectional ports. This allows simultaneous communication to eastern neighbors that takes at most three hops and, therefore, 3 time units through the three bidirectional stages interconnecting the eight nodes in each of the eight groups of nodes. The northern neighbor for each node resides in the adjacent group of eight nodes, which requires five hops, or 5 time units. Thus, the total time required on the fat tree is 8 time units. The fat tree provides full bisection bandwidth, so in the worst case of half the traffic needing to cross the bisection, an effective bandwidth of 56 BW units (as limited by the communication pattern and not by the fattree network) is achieved when packets are continually injected.

The above example should not lead one to the wrong conclusion that meshes are just as good as tori, hypercubes, fat trees, and other networks with higher bisection bandwidth. A number of simplifications that benefit low-bisection networks were assumed to ease the analysis. In practice, packets typically are larger than the link width and occupy links for many more than just one network cycle. Also, many communication patterns do not map so cleanly to the 2D mesh network topology; instead, usually they are more global and irregular in nature. These and other factors combine to increase the chances of packets blocking in low-bisection networks, increasing latency and reducing effective bandwidth.

To put this discussion on topologies into further perspective, Figure F.16 lists various attributes of topologies used in commercial high-performance computers.

F.5

Network Routing, Arbitration, and Switching

Routing, arbitration, and switching are performed at every switch along a packet's path in a switched media network, no matter what the network topology. Numerous interesting techniques for accomplishing these network functions have been

Company	System [network] name	Max. number of nodes [\times # CPUs]	Basic network topology	Injection [reception] node BW in MB/sec	# of data bits per link per direction	Raw network link BW per direction in MB/sec	Raw network bisection BW (bidirectional) in GB/sec
Intel	ASCI Red Paragon	4816 [$\times 2$]	2D mesh 64×64	400 [400]	16 bits	400	51.2
IBM	ASCI White SP Power3 [Colony]	512 [$\times 16$]	Bidirectional MIN with 8-port bidirectional switches (typically a fat tree or Omega)	500 [500]	8 bits (+1 bit of control)	500	256
Intel	Thunder Itanium2 Tiger4 [QsNet ^{II}]	1024 [$\times 4$]	Fat tree with 8-port bidirectional switches	928 [928]	8 bits (+2 of control for 4b/5b encoding)	1333	1365
Cray	XT3 [SeaStar]	30,508 [$\times 1$]	3D torus $40 \times 32 \times 24$	3200 [3200]	12 bits	3800	5836.8
Cray	X1E	1024 [$\times 1$]	4-way bristled 2D torus ($\sim 23 \times 11$) with express links	1600 [1600]	16 bits	1600	51.2
IBM	ASC Purple pSeries 575 [Federation]	>1280 [$\times 8$]	Bidirectional MIN with 8-port bidirectional switches (typically a fat tree or Omega)	2000 [2000]	8 bits (+2 bits of control for novel 5b/6b encoding scheme)	2000	2560
IBM	Blue Gene/L eServer Sol. [Torus Net.]	65,536 [$\times 2$]	3D torus $32 \times 32 \times 64$	612.5 [1050]	1 bit (bit serial)	175	358.4

Figure F.16 Topological characteristics of interconnection networks used in commercial high-performance machines.

proposed in the literature. In this section, we focus on describing a representative set of approaches used in commercial systems for the more commonly used network topologies. Their impact on performance is also highlighted.

Routing

The *routing algorithm* defines which network path, or paths, are allowed for each packet. Ideally, the routing algorithm supplies shortest paths to all packets such that

traffic load is evenly distributed across network links to minimize contention. However, some paths provided by the network topology may not be allowed in order to guarantee that all packets can be delivered, no matter what the traffic behavior. Paths that have an unbounded number of allowed nonminimal hops from packet sources, for instance, may result in packets never reaching their destinations. This situation is referred to as *livelock*. Likewise, paths that cause a set of packets to block in the network forever waiting only for network resources (i.e., links or associated buffers) held by other packets in the set also prevent packets from reaching their destinations. This situation is referred to as *deadlock*. As deadlock arises due to the finiteness of network resources, the probability of its occurrence increases with increased network traffic and decreased availability of network resources. For the network to function properly, the routing algorithm must guard against this anomaly, which can occur in various forms—for example, routing deadlock, request-reply (protocol) deadlock, and fault-induced (reconfiguration) deadlock, etc. At the same time, for the network to provide the highest possible performance, the routing algorithm must be efficient—allowing as many routing options to packets as there are paths provided by the topology, in the best case.

The simplest way of guarding against livelock is to restrict routing such that only minimal paths from sources to destinations are allowed or, less restrictively, only a limited number of nonminimal hops. The strictest form has the added benefit of consuming the minimal amount of network bandwidth, but it prevents packets from being able to use alternative nonminimal paths in case of contention or faults along the shortest (minimal) paths.

Deadlock is more difficult to guard against. Two common strategies are used in practice: avoidance and recovery. In *deadlock avoidance*, the routing algorithm restricts the paths allowed by packets to only those that keep the global network state deadlock-free. A common way of doing this consists of establishing an ordering between a set of resources—the minimal set necessary to support network full access—and granting those resources to packets in some total or partial order such that cyclic dependency cannot form on those resources. This allows an escape path always to be supplied to packets no matter where they are in the network to avoid entering a deadlock state. In *deadlock recovery*, resources are granted to packets without regard for avoiding deadlock. Instead, as deadlock is possible, some mechanism is used to detect the likely existence of deadlock. If detected, one or more packets are removed from resources in the deadlock set—possibly by regressively dropping the packets or by progressively redirecting the packets onto special deadlock recovery resources. The freed network resources are then granted to other packets needing them to resolve the deadlock.

Let us consider routing algorithms designed for distributed switched networks. Figure F.17(a) illustrates one of many possible deadlocked configurations for packets within a region of a 2D mesh network. The routing algorithm can avoid all such deadlocks (and livelocks) by allowing only the use of minimal paths that cross the network dimensions in some total order. That is, links of a given dimension are not supplied to a packet by the routing algorithm until no other links are needed by the packet in all of the preceding dimensions for it to reach its

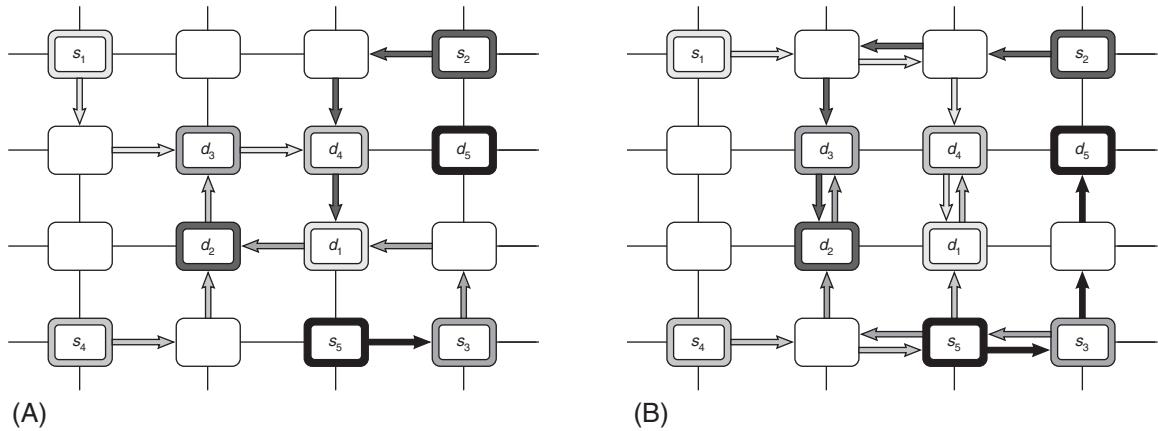


Figure F.17 A mesh network with packets routing from sources, s_i , to destinations, d_i . (a) Deadlock forms from packets destined to d_1 through d_4 blocking on others in the same set that fully occupy their requested buffer resources one hop away from their destinations. This deadlock cycle causes other packets needing those resources also to block, like packets from s_5 destined to d_5 that have reached node s_3 . (b) Deadlock is avoided using dimension-order routing. In this case, packets exhaust their routes in the X dimension before turning into the Y dimension in order to complete their routing.

destination. This is illustrated in Figure F.17(b), where dimensions are crossed in XY dimension order. All the packets must follow the same order when traversing dimensions, exiting a dimension only when links are no longer required in that dimension. This well-known algorithm is referred to as *dimension-order routing* (DOR) or *e-cube routing* in hypercubes. It is used in many commercial systems built from distributed switched networks and on-chip networks. As this routing algorithm always supplies the same path for a given source-destination pair, it is a *deterministic routing* algorithm.

Crossing dimensions in order on some minimal set of resources required to support network full access avoids deadlock in meshes and hypercubes. However, for distributed switched topologies that have wrap-around links (e.g., rings and tori), a total ordering on a minimal set of resources within each dimension is also needed if resources are to be used to full capacity. Alternatively, some empty resources or *bubbles* along the dimensions would be required to remain below full capacity and avoid deadlock. To allow full access, either the physical links must be duplicated or the logical buffers associated with each link must be duplicated, resulting in *physical channels* or *virtual channels*, respectively, on which the ordering is done. Ordering is not necessary on all network resources to avoid deadlock—it is needed only on some minimal set required to support network full access (i.e., some *escape resource set*). Routing algorithms based on this technique (called Duato's protocol) can be defined that allow alternative paths provided by the topology to be used for a given source-destination pair in addition to the escape resource set. One of those allowed paths must be selected, preferably the most

efficient one. Adapting the path in response to prevailing network traffic conditions enables the aggregate network bandwidth to be better utilized and contention to be reduced. Such routing capability is referred to as *adaptive routing* and is used in many commercial systems.

Example How many of the possible dimensional turns are eliminated by dimension-order routing on an n -dimensional mesh network? What is the fewest number of turns that actually need to be eliminated while still maintaining connectedness and deadlock freedom? Explain using a 2D mesh network.

Answer The dimension-order routing algorithm eliminates exactly half of the possible dimensional turns as it is easily proven that all turns from any lower-ordered dimension into any higher-ordered dimension are allowed, but the converse is not true. For example, of the eight possible turns in the 2D mesh shown in Figure F.17, the four turns from $X+$ to $Y+$, $X+$ to $Y-$, $X-$ to $Y+$, and $X-$ to $Y-$ are allowed, where the signs (+ or -) refer to the direction of travel within a dimension. The four turns from $Y+$ to $X+$, $Y+$ to $X-$, $Y-$ to $X+$, and $Y-$ to $X-$ are disallowed turns. The elimination of these turns prevents cycles of any kind from forming—and, thus, avoids deadlock—while keeping the network connected. However, it does so at the expense of not allowing any routing adaptivity.

The *Turn Model* routing algorithm proves that the minimum number of eliminated turns to prevent cycles and maintain connectedness is a quarter of the possible turns, but the right set of turns must be chosen. Only some particular set of eliminated turns allow both requirements to be satisfied. With the elimination of the wrong set of a quarter of the turns, it is possible for combinations of allowed turns to emulate the eliminated ones (and, thus, form cycles and deadlock) or for the network not to be connected. For the 2D mesh, for example, it is possible to eliminate only the two turns ending in the westward direction (i.e., $Y+$ to $X-$ and $Y-$ to $X-$) by requiring packets to start their routes in the westward direction (if needed) to maintain connectedness. Alternatives to this west-first routing for 2D meshes are negative-first routing and north-last routing. For these, the extra quarter of turns beyond that supplied by DOR allows for partial adaptivity in routing, making these adaptive routing algorithms.

Routing algorithms for centralized switched networks can similarly be defined to avoid deadlocks by restricting the use of resources in some total or partial order. For fat trees, resources can be totally ordered along paths starting from the input leaf stage upward to the root and then back down to the output leaf stage. The routing algorithm can allow packets to use resources in increasing partial order, first traversing up the tree until they reach some *least common ancestor* (LCA) of the source and destination, and then back down the tree until they reach their destinations. As there are many least common ancestors for a given destination, multiple alternative paths are allowed while going up the tree, making the routing algorithm adaptive. However, only a single

deterministic path to the destination is provided by the fat tree topology from a least common ancestor. This *self-routing* property is common to many MINs and can be readily exploited: The switch output port at each stage is given simply by shifts of the destination node address.

More generally, a tree graph can be mapped onto any topology—whether direct or indirect—and links between nodes at the same tree level can be allowed by assigning directions to them, where “up” designates paths moving toward the tree root and “down” designates paths moving away from the root node. This allows for generic *up*/down** routing to be defined on any topology such that packets follow paths (possibly adaptively) consisting of zero or more up links followed by zero or more down links to their destination. Up/down ordering prevents cycles from forming, avoiding deadlock. This routing technique was used in Autonet—a self-configuring switched LAN—and in early Myrinet SANs.

Routing algorithms are implemented in practice by a combination of the routing information placed in the packet header by the source node and the routing control mechanism incorporated in the switches. For *source routing*, the entire routing path is precomputed by the source—possibly by table lookup—and placed in the packet header. This usually consists of the output port or ports supplied for each switch along the predetermined path from the source to the destination, which can be stripped off by the routing control mechanism at each switch. An additional bit field can be included in the header to signify whether adaptive routing is allowed (i.e., that any one of the supplied output ports can be used). For *distributed routing*, the routing information usually consists of the destination address. This is used by the routing control mechanism in each switch along the path to determine the next output port, either by computing it using a finite-state machine or by looking it up in a local routing table (i.e., forwarding table). Compared to distributed routing, source routing simplifies the routing control mechanism within the network switches, but it requires more routing bits in the header of each packet, thus increasing the header overhead.

Arbitration

The *arbitration algorithm* determines when requested network paths are available for packets. Ideally, arbiters maximize the matching of free network resources and packets requesting those resources. At the switch level, arbiters maximize the matching of free output ports and packets located in switch input ports requesting those output ports. When all requests cannot be granted simultaneously, switch arbiters resolve conflicts by granting output ports to packets in a fair way such that *starvation* of requested resources by packets is prevented. This could happen to packets in shorter queues if a serve-longest-queue (SLQ) scheme is used. For packets having the same priority level, simple round-robin (RR) or age-based schemes are sufficiently fair and straightforward to implement.

Arbitration can be distributed to avoid centralized bottlenecks. A straightforward technique consists of two phases: a request phase and a grant phase. Let us assume that each switch input port has an associated queue to hold incoming

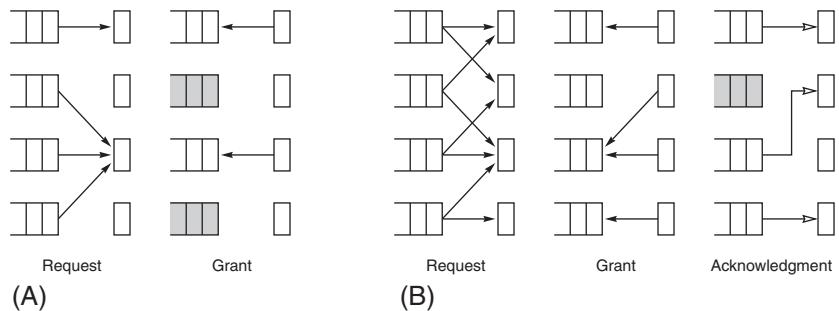


Figure F.18 Two arbitration techniques. (a) Two-phased arbitration in which two of the four input ports are granted requested output ports. (b) Three-phased arbitration in which three of the four input ports are successful in gaining the requested output ports, resulting in higher switch utilization.

packets and that each switch output port has an associated local arbiter implementing a round-robin strategy. Figure F.18(a) shows a possible set of requests for a four-port switch. In the *request phase*, packets at the head of each input port queue send a single request to the arbiters corresponding to the output ports requested by them. Then, each output port arbiter independently arbitrates among the requests it receives, selecting only one. In the *grant phase*, one of the requests to each arbiter is granted the requested output port. When two packets from different input ports request the same output port, only one receives a grant, as shown in the figure. As a consequence, some output port bandwidth remains unused even though all input queues have packets to transmit.

The simple two-phase technique can be improved by allowing several simultaneous requests to be made by each input port, possibly coming from different virtual channels or from multiple adaptive routing options. These requests are sent to different output port arbiters. By submitting more than one request per input port, the probability of matching increases. Now, arbitration requires three phases: request, grant, and acknowledgment. Figure F.18(b) shows the case in which up to two requests can be made by packets at each input port. In the request phase, requests are submitted to output port arbiters, and these arbiters select one of the received requests, as is done for the two-phase arbiter. Likewise, in the grant phase, the selected requests are granted to the corresponding requesters. Taking into account that an input port can submit more than one request, it may receive more than one grant. Thus, it selects among possibly multiple grants using some arbitration strategy such as round-robin. The selected grants are confirmed to the corresponding output port arbiters in the acknowledgment phase.

As can be seen in Figure F.18(b), it could happen that an input port that submits several requests does not receive any grants, while some of the requested ports remain free. Because of this, a second arbitration iteration can improve the probability of matching. In this iteration, only the requests corresponding to non-matched input and output ports are submitted. Iterative arbiters with multiple

requests per input port are able to increase the utilization of switch output ports and, thus, the network link bandwidth. However, this comes at the expense of additional arbiter complexity and increased arbitration delay, which could increase the router clock cycle time if it is on the critical path.

Switching

The *switching technique* defines how connections are established in the network. Ideally, connections between network resources are established or “switched in” only for as long as they are actually needed and exactly at the point that they are ready and needed to be used, considering both time and space. This allows efficient use of available network bandwidth by competing traffic flows and minimal latency. Connections at each hop along the topological path allowed by the routing algorithm and granted by the arbitration algorithm can be established in three basic ways: prior to packet arrival using *circuit switching*, upon receipt of the entire packet using *store-and-forward packet switching*, or upon receipt of only portions of the packet with unit size no smaller than that of the packet header using *cut-through packet switching*.

Circuit switching establishes a circuit *a priori* such that network bandwidth is allocated for packet transmissions along an entire source-destination path. It is possible to pipeline packet transmission across the circuit using staging at each hop along the path, a technique known as *pipelined circuit switching*. As routing, arbitration, and switching are performed only once for one or more packets, routing bits are not needed in the header of packets, thus reducing latency and overhead. This can be very efficient when information is continuously transmitted between devices for the same circuit setup. However, as network bandwidth is removed from the shared pool and preallocated regardless of whether sources are in need of consuming it or not, circuit switching can be very inefficient and highly wasteful of bandwidth.

Packet switching enables network bandwidth to be shared and used more efficiently when packets are transmitted intermittently, which is the more common case. Packet switching comes in two main varieties—store-and-forward and cutthrough switching, both of which allow network link bandwidth to be multiplexed on packet-sized or smaller units of information. This better enables bandwidth sharing by packets originating from different sources. The finer granularity of sharing, however, increases the overhead needed to perform switching: Routing, arbitration, and switching must be performed for every packet, and routing and flow control bits are required for every packet if flow control is used.

Store-and-forward packet switching establishes connections such that a packet is forwarded to the next hop in sequence along its source-destination path only after the entire packet is first stored (staged) at the receiving switch. As packets are completely stored at every switch before being transmitted, links are completely decoupled, allowing full link bandwidth utilization even if links have very different bandwidths. This property is very important in WANs, but the price to pay is packet latency; the total routing, arbitration, and switching delay is multiplicative with the number of hops, as we have seen in Section F.4 when analyzing performance under this assumption.

Cut-through packet switching establishes connections such that a packet can “cut through” switches in a pipelined manner once the header portion of the packet (or equivalent amount of payload trailing the header) is staged at receiving switches. That is, the rest of the packet need not arrive before switching in the granted resources. This allows routing, arbitration, and switching delay to be additive with the number of hops rather than multiplicative to reduce total packet latency. Cut-through comes in two varieties, the main differences being the size of the unit of information on which flow control is applied and, consequently, the buffer requirements at switches. *Virtual cut-through switching* implements flow control at the packet level, whereas *wormhole switching* implements it on flow units, or *flits*, which are smaller than the maximum packet size but usually at least as large as the packet header. Since wormhole switches need to be capable of storing only a small portion of a packet, packets that block in the network may span several switches. This can cause other packets to block on the links they occupy, leading to premature network saturation and reduced effective bandwidth unless some centralized buffer is used within the switch to store them—a technique called *buffered wormhole switching*. As chips can implement relatively large buffers in current technology, virtual cut-through is the more commonly used switching technique. However, wormhole switching may still be preferred in OCNs designed to minimize silicon resources.

Premature network saturation caused by wormhole switching can be mitigated by allowing several packets to share the physical bandwidth of a link simultaneously via time-multiplexed switching at the flit level. This requires physical links to have a set of virtual channels (i.e., the logical buffers mentioned previously) at each end, into which packets are switched. Before, we saw how virtual channels can be used to decouple physical link bandwidth from buffered packets in such a way as to avoid deadlock. Now, virtual channels are multiplexed in such a way that bandwidth is switched in and used by flits of a packet to advance even though the packet may share some links in common with a blocked packet ahead. This, again, allows network bandwidth to be used more efficiently, which, in turn, reduces the average packet latency.

Impact on Network Performance

Routing, arbitration, and switching can impact the packet latency of a loaded network by reducing the contention delay experienced by packets. For an unloaded network that has no contention, the algorithms used to perform routing and arbitration have no impact on latency other than to determine the amount of delay incurred in implementing those functions at switches—typically, the pin-to-pin latency of a switch chip is several tens of nanoseconds. The only change to the best-case packet latency expression given in the previous section comes from the switching technique. Store-and-forward packet switching was assumed before in which transmission delay for the entire packet is incurred on all d hops plus at the source node. For cut-through packet switching, transmission delay is pipelined across the network links comprising the packet’s path at the granularity of the packet header instead of the entire packet. Thus, this delay component is reduced, as shown in the following lower-bound expression for packet latency:

$$\text{Latency} = \text{Sending overhead} + T_{\text{LinkProp}} \times (d + 1) + (T_r + \tau_a + T_s) \times d + \frac{(\text{Packet} + (d \times \text{Header}))}{\text{Bandwidth}} + \text{Receiving overhead}$$

The effective bandwidth is impacted by how efficiently routing, arbitration, and switching allow network bandwidth to be used. The routing algorithm can distribute traffic more evenly across a loaded network to increase the utilization of the aggregate bandwidth provided by the topology—particularly, by the bisection links. The arbitration algorithm can maximize the number of switch output ports that accept packets, which also increases the utilization of network bandwidth. The switching technique can increase the degree of resource sharing by packets, which further increases bandwidth utilization. These combine to affect network bandwidth, BW_{Network} , by an *efficiency factor*, ρ , where $0 < \rho \leq 1$:

$$BW_{\text{Network}} = \rho \times \frac{BW_{\text{Bisection}}}{\gamma}$$

The efficiency factor, ρ , is difficult to calculate or to quantify by means other than simulation. Nevertheless, with this parameter we can estimate the best-case upper-bound effective bandwidth by using the following expression that takes into account the effects of routing, arbitration, and switching:

$$\text{Effective bandwidth} = \min \left(N \times BW_{\text{LinkInjection}}, \rho \times \frac{BW_{\text{Bisection}}}{\gamma}, \sigma \times N \times BW_{\text{LinkReception}} \right)$$

We note that ρ also depends on how well the network handles the traffic generated by applications. For instance, ρ could be higher for circuit switching than for cut-through switching if large streams of packets are continually transmitted between a source-destination pair, whereas the converse could be true if packets are transmitted intermittently.

Example Compare the performance of deterministic routing versus adaptive routing for a 3D torus network interconnecting 4096 nodes. Do so by plotting latency versus applied load and throughput versus applied load. Also compare the efficiency of the best and worst of these networks. Assume that virtual cut-through switching, three-phase arbitration, and virtual channels are implemented. Consider separately the cases for two and four virtual channels, respectively. Assume that one of the virtual channels uses bubble flow control in dimension order so as to avoid deadlock; the other virtual channels are used either in dimension order (for deterministic routing) or minimally along shortest paths (for adaptive routing), as is done in the IBM Blue Gene/L torus network.

Answer It is very difficult to compute analytically the performance of routing algorithms given that their behavior depends on several network design parameters with complex interdependences among them. As a consequence, designers typically resort to cycle-accurate simulators to evaluate performance. One way to evaluate the effect of a certain design decision is to run sets of simulations over a range of network loads, each time modifying one of the design parameters of interest while

keeping the remaining ones fixed. The use of synthetic traffic loads is quite frequent in these evaluations as it allows the network to stabilize at a certain working point and for behavior to be analyzed in detail. This is the method we use here (alternatively, trace-driven or execution-driven simulation can be used).

Figure F.19 shows the typical interconnection network performance plots. On the left, average packet latency (expressed in network cycles) is plotted as a function of applied load (traffic generation rate) for the two routing algorithms with two and four virtual channels each; on the right, throughput (traffic delivery rate) is similarly plotted. Applied load is normalized by dividing it by the number of nodes in the network (i.e., bytes per cycle per node). Simulations are run under the assumption of uniformly distributed traffic consisting of 256-byte packets, where flits are byte sized. Routing, arbitration, and switching delays are assumed to sum to 1 network cycle per hop while the time-of-flight delay over each link is assumed to be 10 cycles. Link bandwidth is 1 byte per cycle, thus providing results that are independent of network clock frequency.

As can be seen, the plots within each graph have similar characteristic shapes, but they have different values. For the latency graph, all start at the no-load latency

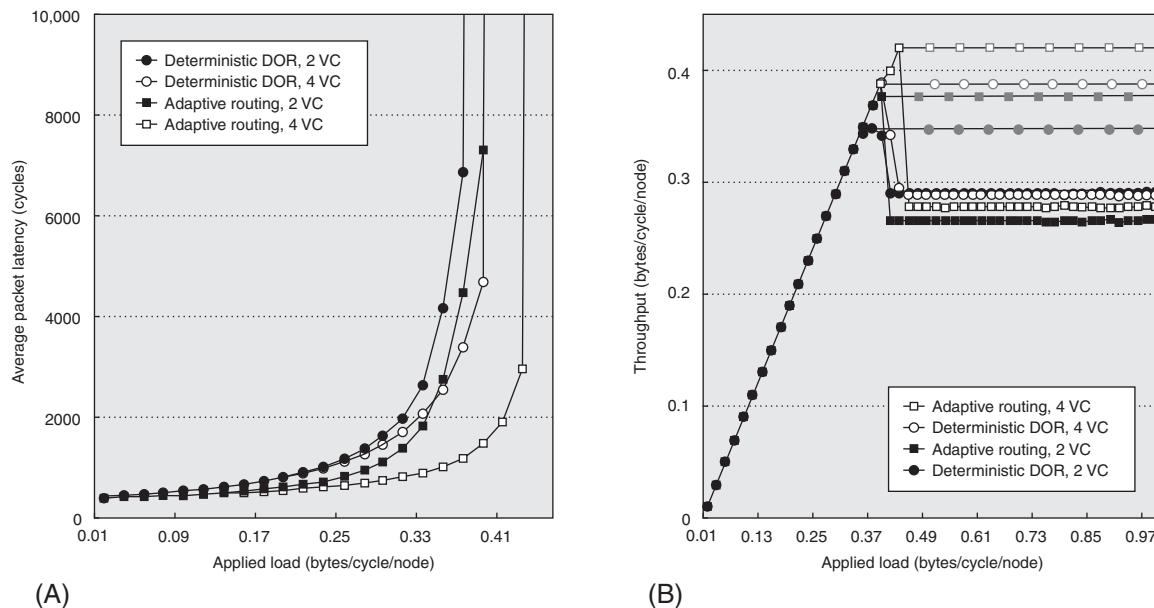


Figure F.19 Deterministic routing is compared against adaptive routing, both with either two or four virtual channels, assuming uniformly distributed traffic on a 4 K node 3D torus network with virtual cut-through switching and bubble flow control to avoid deadlock. (a) Average latency is plotted versus applied load, and (b) throughput is plotted versus applied load (the upper grayish plots show peak throughput, and the lower black plots show sustained throughput). Simulation data were collected by P. Gilabert and J. Flich at the Universidad Politècnica de València, Spain (2006).

as predicted by the latency expression given above, then slightly increase with traffic load as contention for network resources increases. At higher applied loads, latency increases exponentially, and the network approaches its saturation point as it is unable to absorb the applied load, causing packets to queue up at their source nodes awaiting injection. In these simulations, the queues keep growing over time, making latency tend toward infinity. However, in practice, queues reach their capacity and trigger the application to stall further packet generation, or the application throttles itself waiting for acknowledgments/responses to outstanding packets. Nevertheless, latency grows at a slower rate for adaptive routing as alternative paths are provided to packets along congested resources.

For this same reason, adaptive routing allows the network to reach a higher peak throughput for the same number of virtual channels as compared to deterministic routing. At nonsaturation loads, throughput increases fairly linearly with applied load. When the network reaches its saturation point, however, it is unable to deliver traffic at the same rate at which traffic is generated. The saturation point, therefore, indicates the maximum achievable or “peak” throughput, which would be no more than that predicted by the effective bandwidth expression given above. Beyond saturation, throughput tends to drop as a consequence of massive head-of-line blocking across the network (as will be explained further in Section F.6), very much like cars tend to advance more slowly at rush hour. This is an important region of the throughput graph as it shows how significant of a performance drop the routing algorithm can cause if congestion management techniques (discussed briefly in Section F.7) are not used effectively. In this case, adaptive routing has more of a performance drop after saturation than deterministic routing, as measured by the postsaturation sustained throughput.

For both routing algorithms, more virtual channels (i.e., four) give packets a greater ability to pass over blocked packets ahead, allowing for a higher peak throughput as compared to fewer virtual channels (i.e., two). For adaptive routing with four virtual channels, the peak throughput of 0.43 bytes/cycle/node is near the maximum of 0.5 bytes/cycle/node that can be obtained with 100% efficiency (i.e., $\rho = 100\%$), assuming there is enough injection and reception bandwidth to make the network bisection the bottlenecking point. In that case, the network bandwidth is simply 100% times the network bisection bandwidth ($BW_{Bisection}$) divided by the fraction of traffic crossing the bisection (γ), as given by the expression above. Taking into account that the bisection splits the torus into two equally sized halves, γ is equal to 0.5 for uniform traffic as only half the injected traffic is destined to a node at the other side of the bisection. The $BW_{Bisection}$ for a 4096-node 3D torus network is $16 \times 16 \times 4$ unidirectional links times the link bandwidth (i.e., 1 byte/cycle). If we normalize the bisection bandwidth by dividing it by the number of nodes (as we did with network bandwidth), the $BW_{Bisection}$ is 0.25 bytes/cycle/node. Dividing this by γ gives the ideal maximally obtainable network bandwidth of 0.5 bytes/cycle/node.

We can find the efficiency factor, ρ , of the simulated network simply by dividing the measured peak throughput by the ideal throughput. The efficiency factor for

the network with fully adaptive routing and four virtual channels is $0.43/(0.25/0.5) = 86\%$, whereas for the network with deterministic routing and two virtual channels it is $0.37/(0.25/0.5) = 74\%$. Besides the 12% difference in efficiency between the two, another 14% gain in efficiency might be obtained with even better routing, arbitration, switching, and virtual channel designs.

To put this discussion on routing, arbitration, and switching in perspective, Figure F.20 lists the techniques used in SANs designed for commercial high-performance computers. In addition to being applied to the SANs as shown in the figure, the issues discussed in this section also apply to other interconnect domains: from OCNs to WANs.

F.6

Switch Microarchitecture

Network switches implement the routing, arbitration, and switching functions of switched-media networks. Switches also implement buffer management mechanisms and, in the case of lossless networks, the associated flow control. For some networks, switches also implement part of the network management functions that explore, configure, and reconfigure the network topology in response to boot-up and failures. Here, we reveal the internal structure of network switches by describing a basic switch microarchitecture and various alternatives suitable for different routing, arbitration, and switching techniques presented previously.

Basic Switch Microarchitecture

The internal data path of a switch provides connectivity among the input and output ports. Although a shared bus or a multiported central memory could be used, these solutions are insufficient or too expensive, respectively, when the required aggregate switch bandwidth is high. Most high-performance switches implement an internal crossbar to provide nonblocking connectivity within the switch, thus allowing concurrent connections between multiple input-output port pairs. Buffering of blocked packets can be done using first in, first out (FIFO) or circular queues, which can be implemented as *dynamically allocatable multi-queues* (DAMQs) in static RAM to provide high capacity and flexibility. These queues can be placed at input ports (i.e., *input buffered switch*), output ports (i.e., *output buffered switch*), centrally within the switch (i.e., *centrally buffered switch*), or at both the input and output ports of the switch (i.e., *input-output-buffered switch*). Figure F.21 shows a block diagram of an input-output-buffered switch.

Routing can be implemented using a finite-state machine or forwarding table within the routing control unit of switches. In the former case, the routing information given in the packet header is processed by a finite-state machine that determines the allowed switch output port (or ports if routing is adaptive), according to the routing algorithm. Portions of the routing information in the header are usually

Company	System [network] name	Max. number of nodes [\times # CPUs]	Basic network topology	Switch queuing (buffers)	Network routing algorithm	Switch arbitration technique	Network switching technique
Intel	ASCI Red Paragon	4510 [$\times 2$]	2D mesh (64 \times 64)	Input buffered (1 flit)	Distributed dimension-order routing	2-phased RR, distributed across switch	Wormhole with no virtual channels
IBM	ASCI White SP Power3 [Colony]	512 [$\times 16$]	Bidirectional MIN with 8-port bidirectional switches (typically a fat tree or Omega)	Input and central buffer with output queuing (8-way speedup)	Source-based LCA adaptive, shortest-path routing, and table-based multicast routing	2-phased RR, centralized and distributed at outputs for bypass paths	Buffered wormhole and virtual cut-through for multicasting, no virtual channels
Intel	Thunder Itanium2 Tiger4 [QsNet ^{II}]	1024 [$\times 4$]	Fat tree with 8-port bidirectional switches	Input buffered	Source-based LCA adaptive, shortest-path routing	2-phased RR, priority, aging, distributed at output ports	Wormhole with 2 virtual channels
Cray	XT3 [SeaStar]	30,508 [$\times 1$]	3D torus (40 \times 32 \times 24)	Input with staging output	Distributed table-based dimension-order routing	2-phased RR, distributed at output ports	Virtual cut-through with 4 virtual channels
Cray	X1E	1024 [$\times 1$]	4-way bristled 2D torus ($\sim 23 \times 11$) with express links	Input with virtual output queuing	Distributed table-based dimension-order routing	2-phased waveform (pipelined) global arbiter	Virtual cut-through with 4 virtual channels
IBM	ASC Purple pSeries 575 [Federation]	>1280 [$\times 8$]	Bidirectional MIN with 8-port bidirectional switches (typically a fat tree or Omega)	Input and central buffer with output queuing (8-way speedup)	Source and distributed table-based LCA adaptive, shortest-path routing, and multicast	2-phased RR, centralized and distributed at outputs for bypass paths	Buffered wormhole and virtual cut-through for multicasting with 8 virtual channels
IBM	Blue Gene/ L eServer Solution [Torus Net.]	65,536 [$\times 2$]	3D torus (32 \times 32 \times 64)	Input-output buffered	Distributed, adaptive with bubble escape virtual channel	2-phased SLQ, distributed at input and output	Virtual cut-through with 4 virtual channels

Figure F.20 Routing, arbitration, and switching characteristics of interconnections networks in commercial machines.

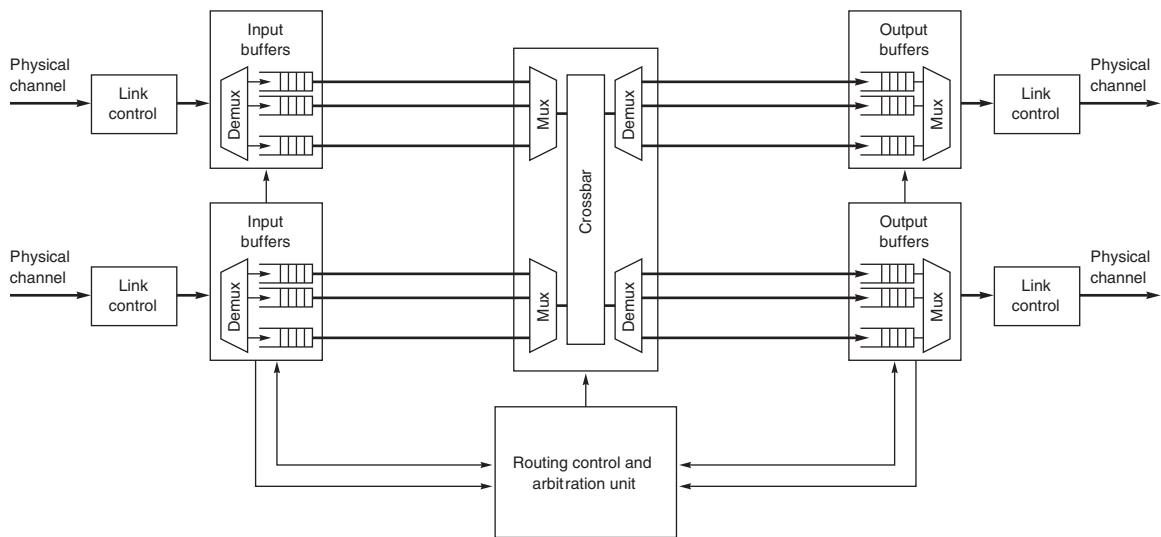


Figure F.21 Basic microarchitectural components of an input-output-buffered switch.

stripped off or modified by the routing control unit after use to simplify processing at the next switch along the path. When routing is implemented using forwarding tables, the routing information given in the packet header is used as an address to access a forwarding table entry that contains the allowed switch output port(s) provided by the routing algorithm. Forwarding tables must be preloaded into the switches at the outset of network operation. Hybrid approaches also exist where the forwarding table is reduced to a small set of routing bits and combined with a small logic block. Those routing bits are used by the routing control unit to know what paths are allowed and decide the output ports the packets need to take. The goal with those approaches is to build flexible yet compact routing control units, eliminating the area and power wastage of a large forwarding table and thus being suitable for OCNs. The routing control unit is usually implemented as a centralized resource, although it could be replicated at every input port so as not to become a bottleneck. Routing is done only once for every packet, and packets typically are large enough to take several cycles to flow through the switch, so a centralized routing control unit rarely becomes a bottleneck. Figure F.21 assumes a centralized routing control unit within the switch.

Arbitration is required when two or more packets concurrently request the same output port, as described in the previous section. Switch arbitration can be implemented in a centralized or distributed way. In the former case, all of the requests and status information are transmitted to the central switch arbitration unit; in the latter case, the arbiter is distributed across the switch, usually among the input and/or output ports. Arbitration may be performed multiple times on packets, and there may be multiple queues associated with each input port,

increasing the number of arbitration requests that must be processed. Thus, many implementations use a hierarchical arbitration approach, where arbitration is first performed locally at every input port to select just one request among the corresponding packets and queues, and later arbitration is performed globally to process the requests made by each of the local input port arbiters. Figure F.21 assumes a centralized arbitration unit within the switch.

The basic switch microarchitecture depicted in Figure F.21 functions in the following way. When a packet starts to arrive at a switch input port, the link controller decodes the incoming signal and generates a sequence of bits, possibly deserializing data to adapt them to the width of the internal data path if different from the external link width. Information is also extracted from the packet header or link control signals to determine the queue to which the packet should be buffered. As the packet is being received and buffered (or after the entire packet has been buffered, depending on the switching technique), the header is sent to the routing unit. This unit supplies a request for one or more output ports to the arbitration unit. Arbitration for the requested output port succeeds if the port is free and has enough space to buffer the entire packet or flit, depending on the switching technique. If wormhole switching with virtual channels is implemented, additional arbitration and allocation steps may be required for the transmission of each individual flit. Once the resources are allocated, the packet is transferred across the internal crossbar to the corresponding output buffer and link if no other packets are ahead of it and the link is free. Link-level flow control implemented by the link controller prevents input queue overflow at the neighboring switch on the other end of the link. If virtual channel switching is implemented, several packets may be time-multiplexed across the link on a flit-by-flit basis. As the various input and output ports operate independently, several incoming packets may be processed concurrently in the absence of contention.

Buffer Organizations

As mentioned above, queues can be located at the switch input, output, or both sides. Output-buffered switches have the advantage of completely eliminating *head-of-line blocking*. Head-of-line (HOL) blocking occurs when two or more packets are buffered in a queue, and a blocked packet at the head of the queue blocks other packets in the queue that would otherwise be able to advance if they were at the queue head. This cannot occur in output-buffered switches as all the packets in a given queue have the same status; they require the same output port. However, it may be the case that all the switch input ports simultaneously receive a packet for the same output port. As there are no buffers at the input side, output buffers must be able to store all those incoming packets at the same time. This requires implementing output queues with an internal switch *speedup* of k . That is, output queues must have a write bandwidth k times the link bandwidth, where k is the number of switch ports. This oftentimes is too expensive. Hence, this solution by itself has rarely been implemented in lossless networks. As the probability of concurrently receiving many packets for the same output port is usually small,

commercial systems that use output-buffered switches typically implement only moderate switch speedup, dropping packets on rare buffer overflow.

Switches with buffers on the input side are able to receive packets without having any switch speedup; however, HOL blocking can occur within input port queues, as illustrated in Figure F.22(a). This can reduce switch output port utilization to less than 60% even when packet destinations are uniformly distributed. As shown in Figure F.22(b), the use of virtual channels (two in this case) can mitigate HOL blocking but does not eliminate it. A more effective solution is to organize the input queues as *virtual output queues* (VOQs), shown in Figure F.22(c). With this, each input port implements as many queues as there are output ports, thus providing separate buffers for packets destined to different output ports. This is a popular technique widely used in ATM switches and IP routers. The main drawbacks of

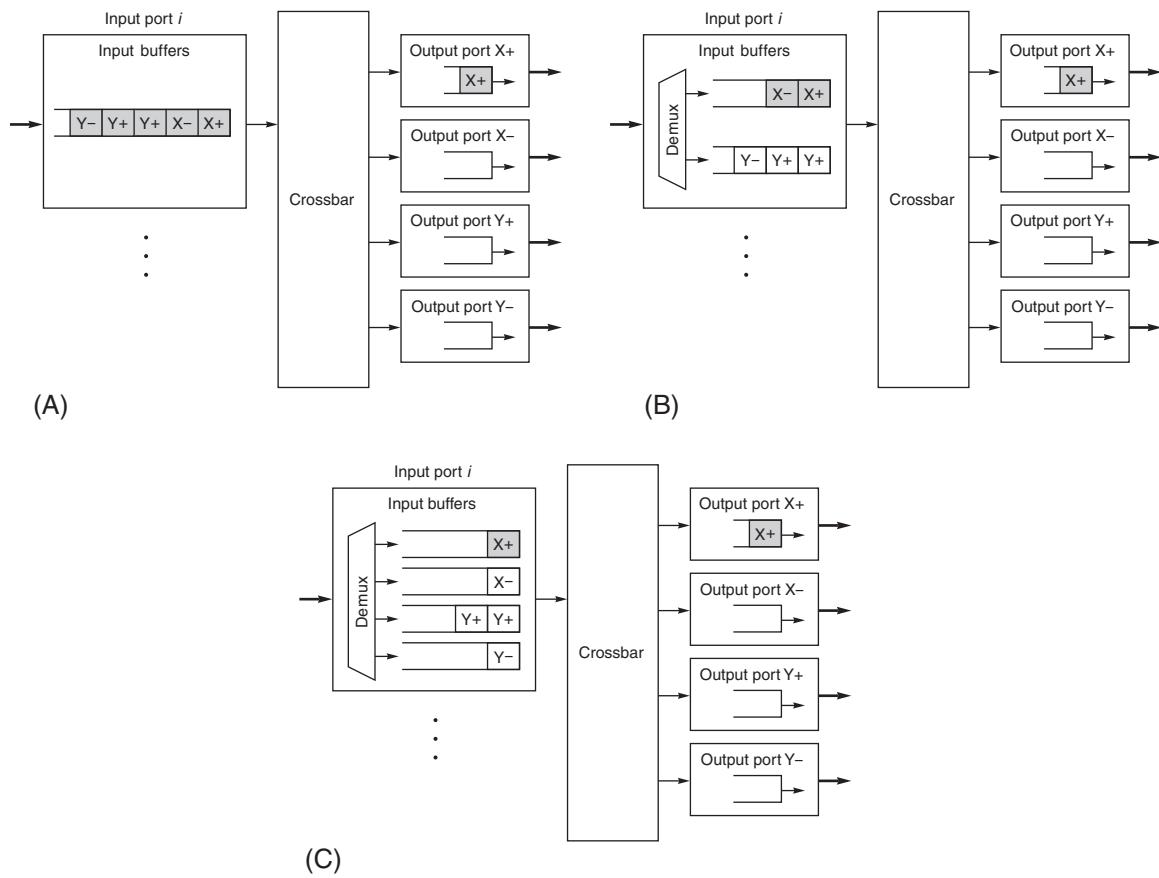


Figure F.22 (a) Head-of-line blocking in an input buffer, (b) the use of two virtual channels to reduce HOL blocking, and (c) the use of virtual output queuing to eliminate HOL blocking within a switch. The shaded input buffer is the one to which the crossbar is currently allocated. This assumes each input port has only one access port to the switch's internal crossbar.

VOQs, however, are cost and lack of scalability: The number of VOQs grows quadratically with switch ports. Moreover, although VOQs eliminate HOL blocking within a switch, HOL blocking occurring at the network level end-to-end is not solved. Of course, it is possible to design a switch with VOQ support at the network level also—that is, to implement as many queues per switch input port as there are output ports across the entire network—but this is extremely expensive. An alternative is to dynamically assign only a fraction of the queues to store (cache) separately only those packets headed for congested destinations.

Combined input-output-buffered switches minimize HOL blocking when there is sufficient buffer space at the output side to buffer packets, and they minimize the switch speedup required due to buffers being at the input side. This solution has the further benefit of decoupling packet transmission through the internal crossbar of the switch from transmission through the external links. This is especially useful for cut-through switching implementations that use virtual channels, where flit transmissions are time-multiplexed over the links. Many designs used in commercial systems implement input-output-buffered switches.

Routing Algorithm Implementation

It is important to distinguish between the routing algorithm and its implementation. While the routing algorithm describes the rules to forward packets across the network and affects packet latency and network throughput, its implementation affects the delay suffered by packets when reaching a node, the required silicon area, and the power consumption associated with the routing computation. Several techniques have been proposed to pre-compute the routing algorithm and/or hide the routing computation delay. However, significantly less effort has been devoted to reduce silicon area and power consumption without significantly affecting routing flexibility. Both issues have become very important, particularly for OCNs. Many existing designs address these issues by implementing relatively simple routing algorithms, but more sophisticated routing algorithms will likely be needed in the future to deal with increasing manufacturing defects, process variability, and other complications arising from continued technology scaling, as discussed briefly below.

As mentioned in a previous section, depending on where the routing algorithm is computed, two basic forms of routing exist: source and distributed routing. In source routing, the complexity of implementation is moved to the end nodes where paths need to be stored in tables, and the path for a given packet is selected based on the destination end node identifier. In distributed routing, however, the complexity is moved to the switches where, at each hop along the path of a packet, a selection of the output port to take is performed. In distributed routing, two basic implementations exist. The first one consists of using a logic block that implements a fixed routing algorithm for a particular topology. The most common example of such an implementation is dimension-order routing, where dimensions are offset in an established order. Alternatively, distributed routing can be implemented with forwarding tables, where each entry encodes the output port to be used for a particular

destination. Therefore, in the worst case, as many entries as destination nodes are required.

Both methods for implementing distributed routing have their benefits and drawbacks. Logic-based routing features a very short computation delay, usually requires a small silicon area, and has low power consumption. However, logic-based routing needs to be designed with a specific topology in mind and, therefore, is restricted to that topology. Table-based distributed routing is quite flexible and supports any topology and routing algorithm. Simply, tables need to be filled with the proper contents based on the applied routing algorithm (e.g., the up*/down* routing algorithm can be defined for any irregular topology). However, the down side of table-based distributed routing is its non-negligible area and power cost. Also, scalability is problematic in table-based solutions as, in the worst case, a system with N end nodes (and switches) requires as many as N tables each with N entries, thus having quadratic cost.

Depending on the network domain, one solution is more suitable than the other. For instance, in SANs, it is usual to find table-based solutions as is the case with InfiniBand. In other environments, like OCNs, table-based implementations are avoided due to the aforementioned costs in power and silicon area. In such environments, it is more advisable to rely on logic-based implementations. Herein lies some of the challenges OCN designers face: ever continuing technology scaling through device miniaturization leads to increases in the number of manufacturing defects, higher failure rates (either transient or permanent), significant process variations (transistors behaving differently from design specs), the need for different clock frequency and voltage domains, and tight power and energy budgets. All of these challenges translate to the network needing support for heterogeneity. Different—possibly irregular—regions of the network will be created owing to failed components, powered down switches and links, disabled components (due to unacceptable variations in performance) and so on. Hence, heterogeneous systems may emerge from a homogeneous design. In this framework, it is important to efficiently implement routing algorithms designed to provide enough flexibility to address these new challenges.

A well-known solution for providing a certain degree of flexibility while being much more compact than traditional table-based approaches is interval routing [Leeuwen 1987], where a range of destinations is defined for each output port. Although this approach is not flexible enough, it provides a clue on how to address emerging challenges. A more recent approach provides a plausible implementation design point that lies between logic-based implementation (efficiency) and table-based implementation (flexibility). Logic-Based Distributed Routing (LBDR) is a hybrid approach that takes as a reference a regular 2D mesh but allows an irregular network to be derived from it due to changes in topology induced by manufacturing defects, failures, and other anomalies. Due to the faulty, disabled, and powered-down components, regularity is compromised and the dimension-order routing algorithm can no longer be used. To support such topologies, LBDR defines a set of configuration bits at each switch. Four connectivity bits are used at each switch to indicate the connectivity of the switch to the neighbor switches in the

topology. Thus, one connectivity bit per port is used. Those connectivity bits are used, for instance, to disable an output port leading to a faulty component. Additionally, eight routing bits are used, two per output port, to define the available routing options. The value of the routing bits is set at power-on and is computed from the routing algorithm to be implemented in the network. Basically, when a routing bit is set, it indicates that a packet can leave the switch through the associated output port and is allowed to perform a certain turn at the next switch. In this respect, LBDR is similar to interval routing, but it defines geographical areas instead of ranges of destinations. Figure F.23 shows an example where a topology-agnostic routing algorithm is implemented with LBDR on an irregular topology. The figure shows the computed configuration bits.

The connectivity and routing bits are used to implement the routing algorithm. For that purpose, a small set of logic gates are used in combination with the configuration bits. Basically, the LBDR approach takes as a reference the initial topology (a 2D mesh), and makes a decision based on the current coordinates of the router, the coordinates of the destination router, and the configuration bits. Figure F.24 shows the required logic, and Figure F.25 shows an example of where a packet is forwarded from its source to its destination with the use of the configuration bits. As can be noticed, routing restrictions are enforced by preventing the use of the west port at switch 10.

LBDR represents a method for efficient routing implementation in OCNs. This mechanism has been recently extended to support non-minimal paths, collective communication operations, and traffic isolation. All of these improvements have been made while maintaining a compact and efficient implementation with the use of a small set of configuration bits. A detailed description of LBDR and its extensions, and the current research on OCNs can be found in Flich [2010].

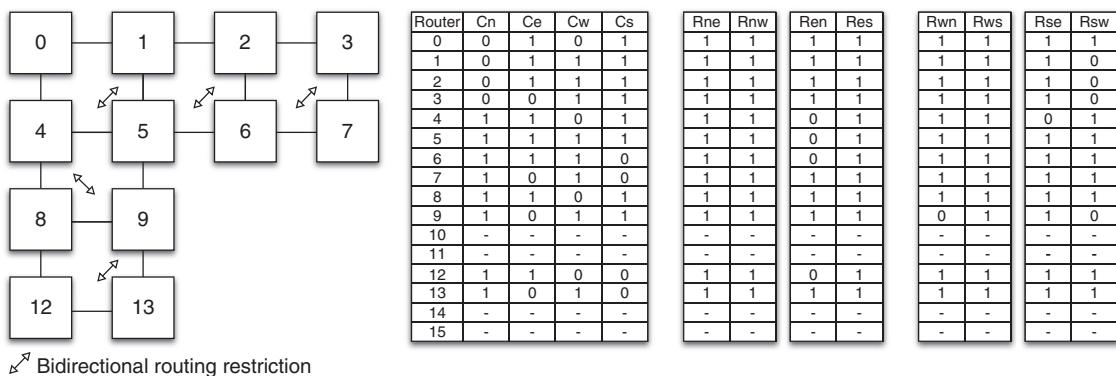


Figure F.23 Shown is an example of an irregular network that uses LBDR to implement the routing algorithm. For each router, connectivity and routing bits are defined.

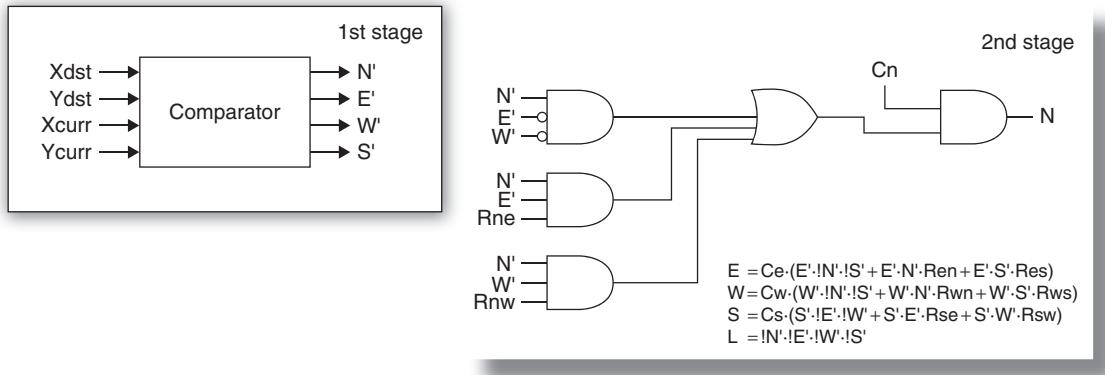


Figure F.24 LBDR logic at each input port of the router.

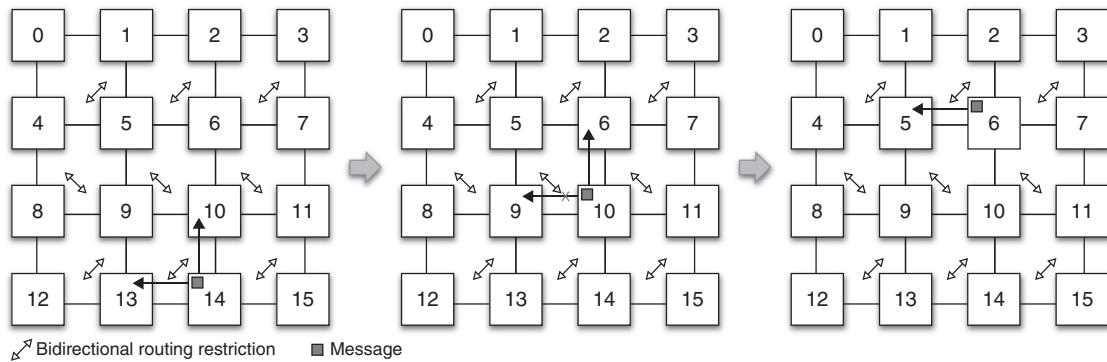


Figure F.25 Example of routing a message from Router 14 to Router 5 using LBDR at each router.

Pipelining the Switch Microarchitecture

Performance can be enhanced by pipelining the switch microarchitecture. Pipelined processing of packets in a switch has similarities with pipelined execution of instructions in a vector processor. In a vector pipeline, a single instruction indicates what operation to apply to all the vector elements executed in a pipelined way. Similarly, in a switch pipeline, a single packet header indicates how to process all of the internal data path physical transfer units (or *phits*) of a packet, which are processed in a pipelined fashion. Also, as packets at different input ports are independent of each other, they can be processed in parallel similar to the way multiple independent instructions or threads of pipelined instructions can be executed in parallel.

The switch microarchitecture can be pipelined by analyzing the basic functions performed within the switch and organizing them into several stages. Figure F.26 shows a block diagram of a five-stage pipelined organization for the basic switch microarchitecture given in Figure F.21, assuming cut-through switching and the use of a forwarding table to implement routing. After receiving the header portion of the packet in the first stage, the routing information (i.e., destination address) is used in the second stage to look up the allowed routing option(s) in the forwarding table. Concurrent with this, other portions of the packet are received and buffered in the input port queue at the first stage. Arbitration is performed in the third stage. The crossbar is configured to allocate the granted output port for the packet in the fourth stage, and the packet header is buffered in the switch output port and ready for transmission over the external link in the fifth stage. Note that the second and

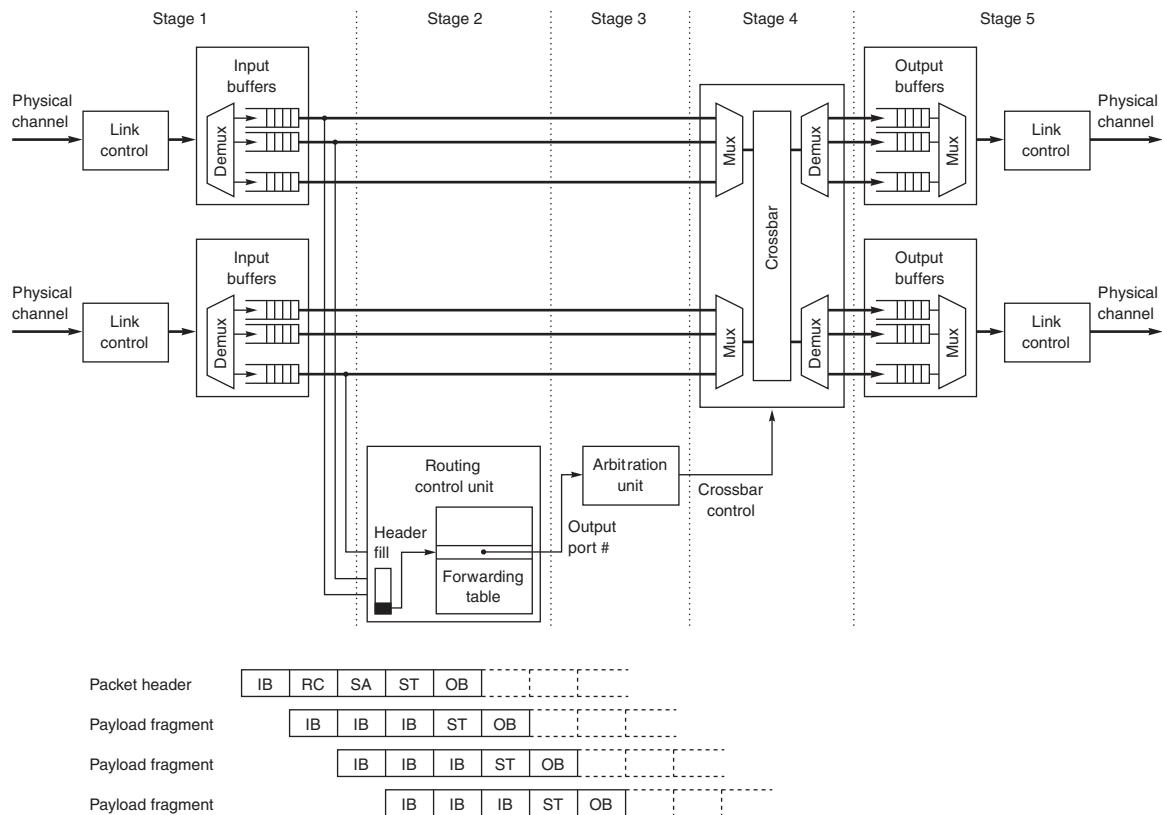


Figure F.26 Pipelined version of the basic input-output-buffered switch. The notation in the figure is as follows: IB is the input link control and buffer stage, RC is the route computation stage, SA is the crossbar switch arbitration stage, ST is the crossbar switch traversal stage, and OB is the output buffer and link control stage. Packet fragments (flits) coming after the header remain in the IB stage until the header is processed and the crossbar switch resources are provided.

third stages are used only by the packet header; the payload and trailer portions of the packet use only three of the stages—those used for data flow-thru once the internal data path of the switch is set up.

A virtual channel switch usually requires an additional stage for virtual channel allocation. Moreover, arbitration is required for every flit before transmission through the crossbar. Finally, depending on the complexity of the routing and arbitration algorithms, several clock cycles may be required for these operations.

Other Switch Microarchitecture Enhancements

As mentioned earlier, internal switch speedup is sometimes implemented to increase switch output port utilization. This speedup is usually implemented by increasing the clock frequency and/or the internal data path width (i.e., phit size) of the switch. An alternative solution consists of implementing several parallel data paths from each input port's set of queues to the output ports. One way of doing this is by increasing the number of crossbar input ports. When implementing several physical queues per input port, this can be achieved by devoting a separate crossbar port to each input queue. For example, the IBM Blue Gene/L implements two crossbar access ports and two read ports per switch input port.

Another way of implementing parallel data paths between input and output ports is to move the buffers to the crossbar crosspoints. This switch architecture is usually referred to as a *buffered crossbar switch*. A buffered crossbar provides independent data paths from each input port to the different output ports, thus making it possible to send up to k packets at a time from a given input port to k different output ports. By implementing independent crosspoint memories for each input-output port pair, HOL blocking is eliminated at the switch level. Moreover, arbitration is significantly simpler than in other switch architectures. Effectively, each output port can receive packets from only a disjoint subset of the crosspoint memories. Thus, a completely independent arbiter can be implemented at each switch output port, each of those arbiters being very simple.

A buffered crossbar would be the ideal switch architecture if it were not so expensive. The number of crosspoint memories increases quadratically with the number of switch ports, dramatically increasing its cost and reducing its scalability with respect to the basic switch architecture. In addition, each crosspoint memory must be large enough to efficiently implement link-level flow control. To reduce cost, most designers prefer input-buffered or combined input-output-buffered switches enhanced with some of the mechanisms described previously.

F.7

Practical Issues for Commercial Interconnection Networks

There are practical issues in addition to the technical issues described thus far that are important considerations for interconnection networks within certain domains. We mention a few of these below.

Connectivity

The type and number of devices that communicate and their communication requirements affect the complexity of the interconnection network and its protocols. The protocols must target the largest network size and handle the types of anomalous systemwide events that might occur. Among some of the issues are the following: How lightweight should the network interface hardware/software be? Should it attach to the memory network or the I/O network? Should it support cache coherence? If the operating system must get involved for every network transaction, the sending and receiving overhead becomes quite large. If the network interface attaches to the I/O network (PCI-Express or HyperTransport interconnect), the injection and reception bandwidth will be limited to that of the I/O network. This is the case for the Cray XT3 SeaStar, Intel Thunder Tiger 4 QsNet^{II}, and many other supercomputer and cluster networks. To support coherence, the sender may have to flush the cache before each send, and the receiver may have to flush its cache before each receive to prevent the stale-data problem. Such flushes further increase sending and receiving overhead, often causing the network interface to be the network bottleneck.

Computer systems typically have a multiplicity of interconnects with different functions and cost-performance objectives. For example, processor-memory interconnects usually provide higher bandwidth and lower latency than I/O interconnects and are more likely to support cache coherence, but they are less likely to follow or become standards. Personal computers typically have a processor-memory interconnect and an I/O interconnect (e.g., PCI-X 2.0, PCIe or Hyper-Transport) designed to connect both fast and slow devices (e.g., USB 2.0, Gigabit Ethernet LAN, Firewire 800). The Blue Gene/L supercomputer uses five interconnection networks, only one of which is the 3D torus used for most of the interprocessor application traffic. The others include a tree-based collective communication network for broadcast and multicast; a tree-based barrier network for combining results (scatter, gather); a control network for diagnostics, debugging, and initialization; and a Gigabit Ethernet network for I/O between the nodes and disk. The University of Texas at Austin's TRIPS Edge processor has eight specialized on-chip networks—some with bidirectional channels as wide as 128 bits and some with 168 bits in each direction—to interconnect the 106 heterogeneous tiles composing the two processor cores with L2 on-chip cache. It also has a chip-to-chip switched network to interconnect multiple chips in a multi-processor configuration. Two of the on-chip networks are switched networks: One is used for operand transport and the other is used for on-chip memory communication. The others are essentially fan-out trees or recombination dedicated link networks used for status and control. The portion of chip area allocated to the interconnect is substantial, with five of the seven metal layers used for global network wiring.

Standardization: Cross-Company Interoperability

Standards are useful in many places in computer design, including interconnection networks. Advantages of successful standards include low cost and stability.

The customer has many vendors to choose from, which keeps price close to cost due to competition. It makes the viability of the interconnection independent of the stability of a single company. Components designed for a standard interconnection may also have a larger market, and this higher volume can reduce the vendors' costs, further benefiting the customer. Finally, a standard allows many companies to build products with interfaces to the standard, so the customer does not have to wait for a single company to develop interfaces to all the products of interest.

One drawback of standards is the time it takes for committees and special-interest groups to agree on the definition of standards, which is a problem when technology is changing rapidly. Another problem is *when* to standardize: On the one hand, designers would like to have a standard before anything is built; on the other hand, it would be better if something were built before standardization to avoid legislating useless features or omitting important ones. When done too early, it is often done entirely by committee, which is like asking all of the chefs in France to prepare a single dish of food—masterpieces are rarely served. Standards can also suppress innovation at that level, since standards fix the interfaces—at least until the next version of the standards surface, which can be every few years or longer. More often, we are seeing consortiums of companies getting together to define and agree on technology that serve as “*de facto*” industry standards. This was the case for InfiniBand.

LANs and WANs use standards and interoperate effectively. WANs involve many types of companies and must connect to many brands of computers, so it is difficult to imagine a proprietary WAN ever being successful. The ubiquitous nature of the Ethernet shows the popularity of standards for LANs as well as WANs, and it seems unlikely that many customers would tie the viability of their LAN to the stability of a single company. Some SANs are standardized such as Fibre Channel, but most are proprietary. OCNs for the most part are proprietary designs, with a few gaining widespread commercial use in system-on-chip (SoC) applications, such as IBM’s CoreConnect and ARM’s AMBA.

Congestion Management

Congestion arises when too many packets try to use the same link or set of links. This leads to a situation in which the bandwidth required exceeds the bandwidth supplied. Congestion by itself does not degrade network performance: simply, the congested links are running at their maximum capacity. Performance degradation occurs in the presence of HOL blocking where, as a consequence of packets going to noncongested destinations getting blocked by packets going to congested destinations, some link bandwidth is wasted and network throughput drops, as illustrated in the example given at the end of Section F.4. *Congestion control* refers to schemes that reduce traffic when the collective traffic of all nodes is too large for the network to handle.

One advantage of a circuit-switched network is that, once a circuit is established, it ensures that there is sufficient bandwidth to deliver all the information

sent along that circuit. Interconnection bandwidth is reserved as circuits are established, and if the network is full, no more circuits can be established. Other switching techniques generally do not reserve interconnect bandwidth in advance, so the interconnection network can become clogged with too many packets. Just as with poor rush-hour commuters, a traffic jam of packets increases packet latency and, in extreme cases, fewer packets per second get delivered by the interconnect. In order to handle congestion in packet-switched networks, some form of *congestion management* must be implemented. The two kinds of mechanisms used are those that control congestion and those that eliminate the performance degradation introduced by congestion.

There are three basic schemes used for congestion control in interconnection networks, each with its own weaknesses: packet discarding, flow control, and choke packets. The simplest scheme is *packet discarding*, which we discussed briefly in Section F.2. If a packet arrives at a switch and there is no room in the buffer, the packet is discarded. This scheme relies on higher-level software that handles errors in transmission to resend lost packets. This leads to significant bandwidth wastage due to (re)transmitted packets that are later discarded and, therefore, is typically used only in lossy networks like the Internet.

The second scheme relies on *flow control*, also discussed previously. When buffers become full, link-level flow control provides feedback that prevents the transmission of additional packets. This *backpressure* feedback rapidly propagates backward until it reaches the sender(s) of the packets producing congestion, forcing a reduction in the injection rate of packets into the network. The main drawbacks of this scheme are that sources become aware of congestion too late when the network is already congested, and nothing is done to alleviate congestion. Back-pressure flow control is common in lossless networks like SANs used in supercomputers and enterprise systems.

A more elaborate way of using flow control is by implementing it directly between the sender and the receiver end nodes, generically called *end-to-end flow control*. *Windowing* is one version of end-to-end credit-based flow control where the window size should be large enough to efficiently pipeline packets through the network. The goal of the window is to limit the number of unacknowledged packets, thus bounding the contribution of each source to congestion, should it arise. The TCP protocol uses a sliding window. Note that end-to-end flow control describes the interaction between just two nodes of the interconnection network, not the entire interconnection network between all end nodes. Hence, flow control helps congestion control, but it is not a global solution.

Choke packets are used in the third scheme, which is built upon the premise that traffic injection should be throttled only when congestion exists across the network. The idea is for each switch to see how busy it is and to enter into a warning state when it passes a threshold. Each packet received by a switch in the warning state is sent back to the source via a choke packet that includes the intended destination. The source is expected to reduce traffic to that destination by a fixed percentage. Since it likely will have already sent other packets along that path, the source node waits for all the packets in transit to be returned before acting on

the choke packets. In this scheme, congestion is controlled by reducing the packet injection rate until traffic reduces, just as metering lights that guard on-ramps control the rate of cars entering a freeway. This scheme works efficiently when the feedback delay is short. When congestion notification takes a long time, usually due to long time of flight, this congestion control scheme may become unstable—reacting too slowly or producing oscillations in packet injection rate, both of which lead to poor network bandwidth utilization.

An alternative to congestion control consists of eliminating the negative consequences of congestion. This can be done by eliminating HOL blocking at every switch in the network as discussed previously. Virtual output queues can be used for this purpose; however, it would be necessary to implement as many queues at every switch input port as devices attached to the network. This solution is very expensive, and not scalable at all. Fortunately, it is possible to achieve good results by dynamically assigning a few set-aside queues to store only the congested packets that travel through some hot-spot regions of the network, very much like caches are intended to store only the more frequently accessed memory locations. This strategy is referred to as *regional explicit congestion notification* (RECN).

Fault Tolerance

The probability of system failures increases as transistor integration density and the number of devices in the system increases. Consequently, system reliability and availability have become major concerns and will be even more important in future systems with the proliferation of interconnected devices. A practical issue arises, therefore, as to whether or not the interconnection network relies on all the devices being operational in order for the network to work properly. Since software failures are generally much more frequent than hardware failures, another question surfaces as to whether a software crash on a single device can prevent the rest of the devices from communicating. Although some hardware designers try to build fault-free networks, in practice, it is only a question of the rate of failures, not whether they can be prevented. Thus, the communication subsystem must have mechanisms for dealing with faults when—not if—they occur.

There are two main kinds of failure in an interconnection network: *transient* and *permanent*. Transient failures are usually produced by electromagnetic interference and can be detected and corrected using the techniques described in Section F.2. Oftentimes, these can be dealt with simply by retransmitting the packet either at the link level or end-to-end. Permanent failures occur when some component stops working within specifications. Typically, these are produced by overheating, overbiasing, overuse, aging, and so on and cannot be recovered from simply by retransmitting packets with the help of some higher-layer software protocol. Either an alternative physical path must exist in the network and be supplied by the routing algorithm to circumvent the fault or the network will be crippled, unable to deliver packets whose only paths are through faulty resources.

Three major categories of techniques are used to deal with permanent failures: *resource sparing*, *fault-tolerant routing*, and *network reconfiguration*. In the first

technique, faulty resources are switched off or bypassed, and some spare resources are switched in to replace the faulty ones. As an example, the ServerNet interconnection network is designed with two identical switch fabrics, only one of which is usable at any given time. In case of failure in one fabric, the other is used. This technique can also be implemented without switching in spare resources, leading to a degraded mode of operation after a failure. The IBM Blue Gene/L supercomputer, for instance, has the facility to bypass failed network resources while retaining its base topological structure and routing algorithm. The main drawback of this technique is the relatively large number of healthy resources (e.g., midplane node boards) that may need to be switched off after a failure in order to retain the base topological structure (e.g., a 3D torus).

Fault-tolerant routing, on the other hand, takes advantage of the multiple paths already existing in the network topology to route messages in the presence of failures without requiring spare resources. Alternative paths for each supported fault combination are identified at design time and incorporated into the routing algorithm. When a fault is detected, a suitable alternative path is used. The main difficulty when using this technique is guaranteeing that the routing algorithm will remain deadlock-free when using the alternative paths, given that arbitrary fault patterns may occur. This is especially difficult in direct networks whose regularity can be compromised by the fault pattern. The Cray T3E is an example system that successfully applies this technique on its 3D torus direct network. There are many examples of this technique in systems using indirect networks, such as with the bidirectional multistage networks in the ASCI White and ASC Purple. Those networks provide multiple minimal paths between end nodes and, inherently, have no routing deadlock problems (see Section F.5). In these networks, alternative paths are selected at the source node in case of failure.

Network reconfiguration is yet another, more general technique to handle voluntary and involuntary changes in the network topology due either to failures or to some other cause. In order for the network to be reconfigured, the nonfaulty portions of the topology must first be discovered, followed by computation of the new routing tables and distribution of the routing tables to the corresponding network locations (i.e., switches and/or end node devices). Network reconfiguration requires the use of programmable switches and/or network interfaces, depending on how routing is performed. It may also make use of generic routing algorithms (e.g., up*/down* routing) that can be configured for all the possible network topologies that may result after faults. This strategy relieves the designer from having to supply alternative paths for each possible fault combination at design time. Programmable network components provide a high degree of flexibility but at the expense of higher cost and latency. Most standard and proprietary interconnection networks for clusters and SANs—including Myrinet, Quadrics, InfiniBand, Advanced Switching, and Fibre Channel—incorporate software for (re)configuring the network routing in accordance with the prevailing topology.

Another practical issue ties to node failure tolerance. If an interconnection network can survive a failure, can it also continue operation while a new node is added to or removed from the network, usually referred to as *hot swapping*? If not, each addition or removal of a new node disables the interconnection network, which is

impractical for WANs and LANs and is usually intolerable for most SANs. Online system expansion requires hot swapping, so most networks allow for it. Hot swapping is usually supported by implementing *dynamic network reconfiguration*, in which the network is reconfigured without having to stop user traffic. The main difficulty with this is guaranteeing deadlock-free routing while routing tables for switches and/or end node devices are dynamically and asynchronously updated as more than one routing algorithm may be alive (and, perhaps, clashing) in the network at the same time. Most WANs solve this problem by dropping packets whenever required, but dynamic network reconfiguration is much more complex in lossless networks. Several theories and practical techniques have recently been developed to address this problem efficiently.

Example

Figure F.27 shows the number of failures of 58 desktop computers on a local area network for a period of just over one year. Suppose that one local area network is based on a network that requires all machines to be operational for the interconnection network to send data; if a node crashes, it cannot accept messages, so the interconnection becomes choked with data waiting to be delivered. An alternative is the traditional local area network, which can operate in the presence of node failures; the interconnection simply discards messages for a node that decides not to accept them. Assuming that you need to have both your workstation and the connecting LAN to get your work done, how much greater are your chances of being prevented from getting your work done using the failure-intolerant LAN versus traditional LANs? Assume the downtime for a crash is less than 30 minutes. Calculate using the one-hour intervals from this figure.

Answer

Assuming the numbers for Figure F.27, the percentage of hours that you can't get your work done using the failure-intolerant network is

$$\begin{aligned} \frac{\text{Intervals with failures}}{\text{Total intervals}} &= \frac{\text{Total intervals} - \text{Intervals with no failures}}{\text{Total intervals}} \\ &= \frac{8974 - 8605}{8974} = \frac{369}{8974} = 4.1\% \end{aligned}$$

The percentage of hours that you can't get your work done using the traditional network is just the time your workstation has crashed. If these failures are equally distributed among workstations, the percentage is

$$\frac{\text{Failures/Machines}}{\text{Total intervals}} = \frac{654/58}{8974} = \frac{11.28}{8974} = 0.13\%$$

Hence, you are more than 30 times more likely to be prevented from getting your work done with the failure-intolerant LAN than with the traditional LAN, according to the failure statistics in Figure F.27. Stated alternatively, the person responsible for maintaining the LAN would receive a 30-fold increase in phone calls from irate users!

Failed machines per time interval	One-hour intervals with number of failed machines in first column	Total failures per one-hour interval	One-day intervals with number of failed machines in first column	Total failures per one-day interval
0	8605	0	184	0
1	264	264	105	105
2	50	100	35	70
3	25	75	11	33
4	10	40	6	24
5	7	35	9	45
6	3	18	6	36
7	1	7	4	28
8	1	8	4	32
9	2	18	2	18
10	2	20		
11	1	11	2	22
12			1	12
17	1	17		
20	1	20		
21	1	21	1	21
31			1	31
38			1	38
58			1	58
Total	8974	654	373	573

Figure F.27 Measurement of reboots of 58 DECstation 5000 s running Ultrix over a 373-day period. These reboots are distributed into time intervals of one hour and one day. The first column sorts the intervals according to the number of machines that failed in that interval. The next two columns concern one-hour intervals, and the last two columns concern one-day intervals. The second and fourth columns show the number of intervals for each number of failed machines. The third and fifth columns are just the product of the number of failed machines and the number of intervals. For example, there were 50 occurrences of one-hour intervals with 2 failed machines, for a total of 100 failed machines, and there were 35 days with 2 failed machines, for a total of 70 failures. As we would expect, the number of failures per interval changes with the size of the interval. For example, the day with 31 failures might include one hour with 11 failures and one hour with 20 failures. The last row shows the total number of each column; the number of failures doesn't agree because multiple reboots of the same machine in the same interval do not result in separate entries. (Randy Wang of the University of California–Berkeley collected these data.)

F.8

Examples of Interconnection Networks

To further provide mass to the concepts described in the previous sections, we look at five example networks from the four interconnection network domains considered in this appendix. In addition to one for each of the OCN, LAN, and WAN areas, we look at two examples from the SAN area: one for system area networks

and one for system/storage area networks. The first two examples are proprietary networks used in high-performance systems; the latter three examples are network standards widely used in commercial systems.

On-Chip Network: Intel Single-Chip Cloud Computer

With continued increases in transistor integration as predicted by Moore's law, processor designers are under the gun to find ways of combating chip-crossing wire delay and other problems associated with deep submicron technology scaling. Multicore microarchitectures have gained popularity, given their advantages of simplicity, modularity, and ability to exploit parallelism beyond that which can be achieved through aggressive pipelining and multiple instruction/data issuing on a single core. No matter whether the processor consists of a single core or multiple cores, higher and higher demands are being placed on intrachip communication bandwidth to keep pace—not to mention interchip bandwidth. This has spurred a great amount of interest in OCN designs that efficiently support communication of instructions, register operands, memory, and I/O data within and between processor cores both on and off the chip. Here we focus on one such on-chip network: The Intel Single-chip Cloud Computer prototype.

The Single-chip Cloud Computer (SCC) is a prototype chip multiprocessor with 48 Intel IA-32 architecture cores. Cores are laid out (see Figure F.28) on a network with a 2D mesh topology (6×4). The network connects 24 tiles, 4 on-die memory controllers, a voltage regulator controller (VRC), and an external system interface controller (SIF). In each tile two cores are connected to a router. The four memory controllers are connected at the boundaries of the mesh, two on each side, while the VRC and SIF controllers are connected at the bottom border of the mesh.

Each memory controller can address two DDR3 DIMMS, each up to 8 GB of memory, thus resulting in a maximum of 64 GB of memory. The VRC controller allows any core or the system interface to adjust the voltage in any of the six pre-defined regions configuring the network (two 2-tile regions). The clock can also be adjusted at a finer granularity with each tile having its own operating frequency. These regions can be turned off or scaled down for large power savings. This method allows full application control of the power state of the cores. Indeed, applications have an API available to define the voltage and the frequency of each region. The SIF controller is used to communicate the network from outside the chip.

Each of the tiles includes two processor cores (P54C-based IA) with associated L1 16 KB data cache and 16 KB instruction cache and a 256 KB L2 cache (with the associated controller), a 5-port router, traffic generator (for testing purposes only), a mesh interface unit (MIU) handling all message passing requests, memory look-up tables (with configuration registers to set the mapping of a core's physical addresses to the extended memory map of the system), a message-passing buffer, and circuitry for the clock generation and synchronization for crossing asynchronous boundaries.

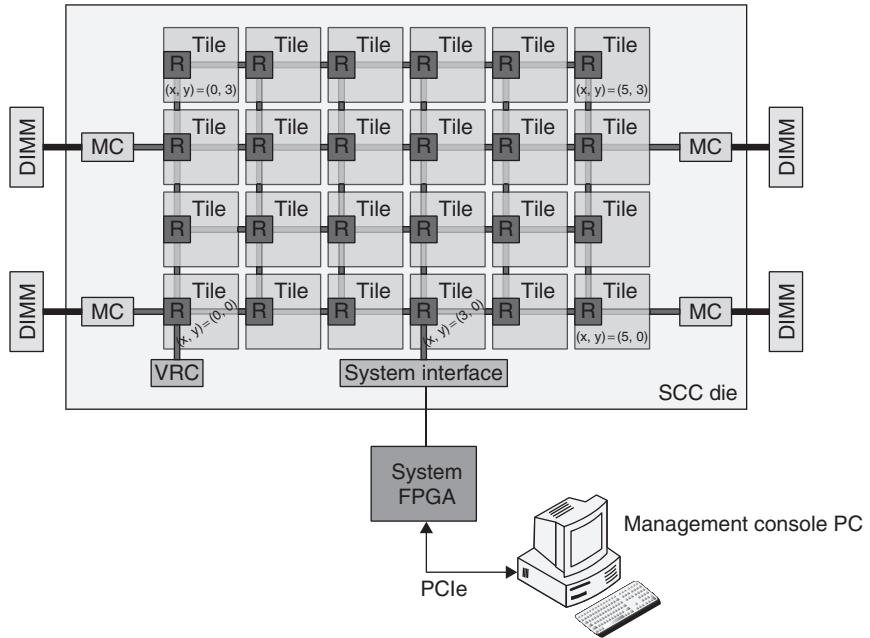


Figure F.28 SCC Top-level architecture. From Howard, J. et al., *IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pp. 58–59.

Focusing on the OCN, the MIU unit is in charge of interfacing the cores to the network, including the packetization and de-packetization of large messages; command translation and address decoding/lookup; link-level flow control and credit management; and arbiter decisions following a round-robin scheme. A credit-based flow control mechanism is used together with virtual cut-through switching (thus making it necessary to split long messages into packets). The routers are connected in a 2D mesh layout, each on its own power supply and clock source. Links connecting routers have 16B + 2B side bands running at 2 GHz. Zero-load latency is set to 4 cycles, including link traversal. Eight virtual channels are used for performance (6 VCs) and protocol-level deadlock handling (2 VCs). A message-level arbitration is implemented by a wrapped wave-front arbiter. The dimension-order XY routing algorithm is used and pre-computation of the output port is performed at every router.

Besides the tiles having regions defined for voltage and frequency, the network (made of routers and links) has its own single region. Thus, all the network components run at the same speed and use the same power supply. An asynchronous clock transition is required between the router and the tile.

One of the distinctive features of the SCC architecture is the support for a messaging-based communication protocol rather than hardware cache-coherent

memory for inter-core communication. Message passing buffers are located on every router and APIs are provided to take full control of MPI structures. Cache coherency can be implemented by software.

The SCC router represents a significant improvement over the Teraflops processor chip in the implementation of a 2D on-chip interconnect. Contrasted with the 2D mesh implemented in the Teraflops processor, this implementation is tuned for a wider data path in a multiprocessor interconnect and is more latency, area, and power optimized for such a width. It targets a lower 2-GHz frequency of operation compared to the 5 GHz of its predecessor Teraflops processor, yet with a higher-performance interconnect architecture.

System Area Network: IBM Blue Gene/L 3D Torus Network

The IBM BlueGene/L was the largest-scaled, highest-performing computer system in the world in 2005, according to www.top500.org. With 65,536 dual-processor compute nodes and 1024 I/O nodes, this 360 TFLOPS (peak) supercomputer has a system footprint of approximately 2500 square feet. Both processors at each node can be used for computation and can handle their own communication protocol processing in virtual mode or, alternatively, one of the processors can be used for computation and the other for network interface processing. Packets range in size from 32 bytes to a maximum of 256 bytes, and 8 bytes are used for the header. The header includes routing, virtual channel, link-level flow control, packet size, and other such information, along with 1 byte for CRC to protect the header. Three bytes are used for CRC at the packet level, and 1 byte serves as a valid indicator.

The main interconnection network is a proprietary $32 \times 32 \times 64$ 3D torus SAN that interconnects all 64 K nodes. Each node switch has six 350 MB/sec bidirectional links to neighboring torus nodes, an injection bandwidth of 612.5 MB/sec from the two node processors, and a reception bandwidth of 1050 MB/sec to the two node processors. The reception bandwidth from the network equals the inbound bandwidth across all switch ports, which prevents reception links from bottlenecking network performance. Multiple packets can be sunk concurrently at each destination node because of the higher reception link bandwidth.

Two nodes are implemented on a $2 \times 1 \times 1$ compute card, 16 compute cards and 2 I/O cards are implemented on a $4 \times 4 \times 2$ node board, 16 node boards are implemented on an $8 \times 8 \times 8$ midplane, and 2 midplanes form a 1024-node rack with physical dimensions of $0.9 \times 0.9 \times 1.9$ cubic meters. Links have a maximum physical length of 8.6 meters, thus enabling efficient link-level flow control with reasonably low buffering requirements. Low latency is achieved by implementing virtual cut-through switching, distributing arbitration at switch input and output ports, and precomputing the current routing path at the previous switch using a finite-state machine so that part of the routing delay is removed from the critical path in switches. High effective bandwidth is achieved using input-buffered

switches with dual read ports, virtual cut-through switching with four virtual channels, and fully adaptive deadlock-free routing based on bubble flow control.

A key feature in networks of this size is fault tolerance. Failure rate is reduced by using a relatively low link clock frequency of 700 MHz (same as processor clock) on which both edges of the clock are used (i.e., 1.4 Gbps or 175 MB/sec transfer rate is supported for each bit-serial network link in each direction), but failures may still occur in the network. In case of failure, the midplane node boards containing the fault(s) are switched off and bypassed to isolate the fault, and computation resumes from the last checkpoint. Bypassing is done using separate bypass switch boards associated with each midplane that are additional to the set of torus node boards. Each bypass switch board can be configured to connect either to the corresponding links in the midplane node boards or to the next bypass board, effectively removing the corresponding set of midplane node boards. Although the number of processing nodes is reduced to some degree in some network dimensions, the machine retains its topological structure and routing algorithm.

Some collective communication operations such as barrier synchronization, broadcast/multicast, reduction, and so on are not performed well on the 3D torus as the network would be flooded with traffic. To remedy this, two separate tree networks with higher per-link bandwidth are used to implement collective and combining operations more efficiently. In addition to providing support for efficient synchronization and broadcast/multicast, hardware is used to perform some arithmetic reduction operations in an efficient way (e.g., to compute the sum or the maximum value of a set of values, one from each processing node). In addition to the 3D torus and the two tree networks, the Blue Gene/L implements an I/O Gigabit Ethernet network and a control system Fast Ethernet network of lower bandwidth to provide for parallel I/O, configuration, debugging, and maintenance.

System/Storage Area Network: InfiniBand

InfiniBand is an industrywide *de facto* networking standard developed in October 2000 by a consortium of companies belonging to the InfiniBand Trade Association. InfiniBand can be used as a system area network for interprocessor communication or as a storage area network for server I/O. It is a switch-based interconnect technology that provides flexibility in the topology, routing algorithm, and arbitration technique implemented by vendors and users. InfiniBand supports data transmission rates of 2 to 120 Gbp/link per direction across distances of 300 meters. It uses cut-through switching, 16 virtual channels and service levels, credit-based link-level flow control, and weighted round-robin fair scheduling and implements programmable forwarding tables. It also includes features useful for increasing reliability and system availability, such as communication subnet management, end-to-end path establishment, and virtual destination naming.

Institution and processor [network] name	Year built	Number of network ports [cores or tiles + other ports]	Basic network topology	# of data bits per link per direction	Link bandwidth [link clock speed]	Routing; arbitration; switching	# of chip metal layers; flow control; #virtual channels
MIT Raw [General Dynamic Network]	2002	16 ports [16 tiles]	2D mesh (4 × 4)	32 bits	0.9 GB/sec [225 MHz, clocked at proc speed]	XY DOR with request-reply deadlock recovery; RR arbitration; wormhole	6 layers; credit-based no virtual channels
IBM Power5	2004	7 ports [2 PE cores + 5 other ports]	Crossbar	256 bits Inst fetch; 64 bits for stores; 256 bits LDs	[1.9 GHz, clocked at proc speed]	Shortest-path; nonblocking; circuit switch	7 layers; handshaking; no virtual channels
U.T. Austin TRIP Edge [Operand Network]	2005	25 ports [25 execution unit tiles]	2D mesh (5 × 5)	110 bits	5.86 GB/sec [533 MHz clock scaled by 80%]	YX DOR; distributed RR arbitration; wormhole	7 layers; on/off flow control; no virtual channels
U.T. Austin TRIP Edge [On-Chip Network]	2005	40 ports [16 L2 tiles + 24 network interface tile]	2D mesh (10 × 4)	128 bits	6.8 GB/sec [533 MHz clock scaled by 80%]	YX DOR; distributed RR arbitration; VCT switched	7 layers; credit-based flow control; 4 virtual channels
Sony, IBM, Toshiba Cell BE [Element Interconnect Bus]	2005	12 ports [1 PPE and 8 SPEs + 3 other ports for memory, I/O interface]	Ring (4 total, 2 in each direction)	128 bits data (+16 bits tag)	25.6 GB/sec [1.6 GHz, clocked at half the proc speed]	Shortest-path; tree-based RR arbitration (centralized); pipelined circuit switch	8 layers; credit-based flow control; no virtual channels
Sun UltraSPARC T1 processor	2005	Up to 13 ports [8 PE cores + 4 L2 banks + 1 shared I/O]	Crossbar	128 bits both for the 8 cores and the 4 L2 banks	19.2 GB/sec [1.2 GHz, clocked at proc speed]	Shortest-path; age-based arbitration; VCT switched	9 layers; handshaking; no virtual channels

Figure F.29 Characteristics of on-chip networks implemented in recent research and commercial processors. Some processors implement multiple on-chip networks (not all shown)—for example, two in the MIT Raw and eight in the TRIP Edge.

Figure F.30 shows the packet format for InfiniBand juxtaposed with two other network standards from the LAN and WAN areas. Figure F.31 compares various characteristics of the InfiniBand standard with two proprietary system area networks widely used in research and commercial high-performance computer systems.

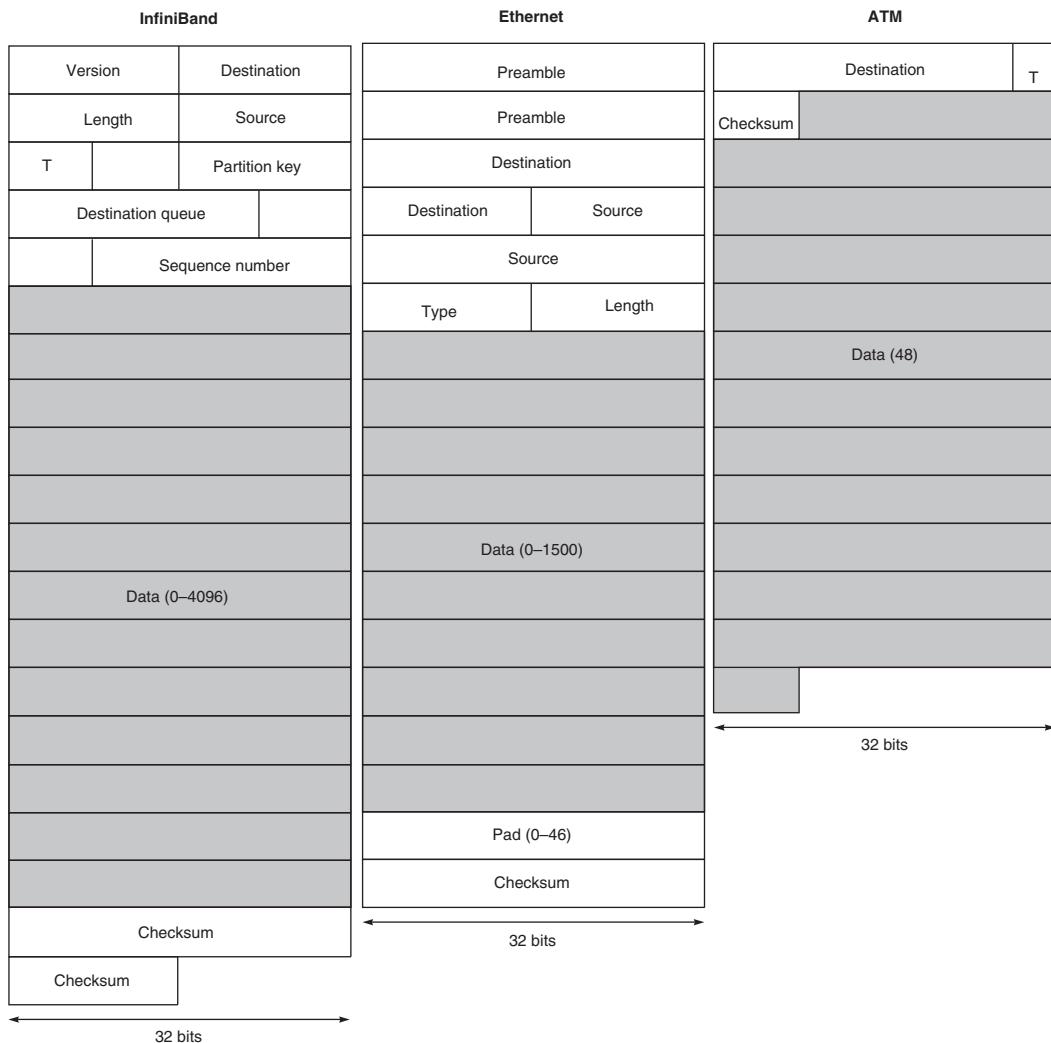


Figure F.30 Packet format for InfiniBand, Ethernet, and ATM. ATM calls their messages “cells” instead of packets, so the proper name is ATM cell format. The width of each drawing is 32 bits. All three formats have destination addressing fields, encoded differently for each situation. All three also have a checksum field to catch transmission errors, although the ATM checksum field is calculated only over the header; ATM relies on higher-level protocols to catch errors in the data. Both InfiniBand and Ethernet have a length field, since the packets hold a variable amount of data, with the former counted in 32-bit words and the latter in bytes. InfiniBand and ATM headers have a type field (T) that gives the type of packet. The remaining Ethernet fields are a preamble to allow the receiver to recover the clock from the self-clocking code used on the Ethernet, the source address, and a pad field to make sure the smallest packet is 64 bytes (including the header). InfiniBand includes a version field for protocol version, a sequence number to allow in-order delivery, a field to select the destination queue, and a partition key field. Infiniband has many more small fields not shown and many other packet formats; above is a simplified view. ATM’s short, fixed packet is a good match to real-time demand of digital voice.

Network name [vendors]	Used in top 10 supercomputer clusters (2005)	Number of nodes	Basic network topology	Raw link bidirectional BW	Routing algorithm	Arbitration technique	Switching technique; flow control
InfiniBand [Mellanox, Voltair]	SGI Altrix and Dell Poweredge Thunderbird	>Millions (2^{128} GUID addresses, like IPv6)	Completely configurable (arbitrary)	4–240 Gbps	Arbitrary (table-driven), typically up*/down*	Weighted RR fair scheduling (2-level priority)	Cut-through, 16 virtual channels (15 for data); credit-based
Myrinet-2000 [Myricom]	Barcelona Supercomputer Center in Spain	8192 nodes	Bidirectional MIN with 16-port bidirectional switches (Clos net.)	4 Gbps	Source-based dispersive (adaptive) minimal routing	Round-robin arbitration	Cut-through switching with no virtual channels; Xon/Xoff flow control
QsNet ^{II} [Quadrics]	Intel Thunder Itanium2 Tiger4	>Tens of thousands	Fat tree with 8-port bidirectional switches	21.3 Gbps	Source-based LCA adaptive shortest-path routing	2-phased RR, priority, aging, distributed at output ports	Wormhole with 2 virtual channels; credit-based

Figure F.31 Characteristics of system area networks implemented in various top 10 supercomputer clusters in 2005.

InfiniBand offers two basic mechanisms to support user-level communication: send/receive and remote DMA (RDMA). With send/receive, the receiver has to explicitly post a receive buffer (i.e., allocate space in its channel adapter network interface) before the sender can transmit data. With RDMA, the sender can remotely DMA data directly into the receiver device's memory. For example, for a nominal packet size of 4 bytes measured on a Mellanox MHEA28-XT channel adapter connected to a 3.4 GHz Intel Xeon host device, sending and receiving overhead is 0.946 and 1.423 μ s, respectively, for the send/receive mechanism, whereas it is 0.910 and 0.323 μ s, respectively, for the RDMA mechanism.

As discussed in Section F.2, the packet size is important in getting full benefit of the network bandwidth. One might ask, "What is the natural size of messages?" Figure F.32(a) shows the size of messages for a commercial fluid dynamics simulation application, called Fluent, collected on an InfiniBand network at The Ohio State University's Network-Based Computer Laboratory. One plot is cumulative in messages sent and the other is cumulative in data bytes sent. Messages in this graph are message passing interface (MPI) units of information, which gets divided into InfiniBand maximum transfer units (packets) transferred over the network. As shown, the maximum message size is over 512 KB, but approximately 90% of the messages are less than 512 bytes. Messages of 2 KB represent approximately 50% of the bytes transferred. An Integer Sort application kernel in the NAS Parallel

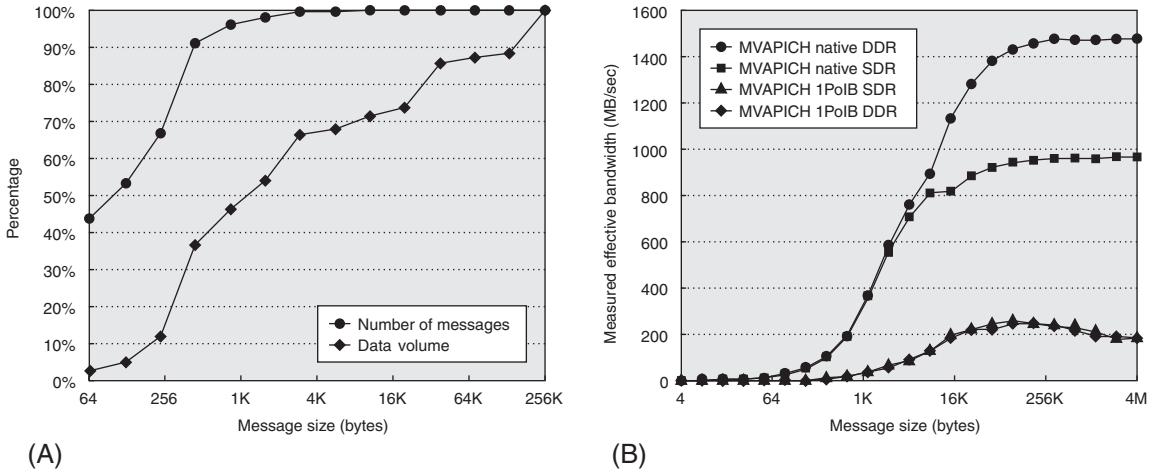


Figure F.32 Data collected by D.K. Panda, S. Sur, and L. Chai (2005) in the Network-Based Computing Laboratory at The Ohio State University. (a) Cumulative percentage of messages and volume of data transferred as message size varies for the Fluent application (www.fluent.com). Each x-axis entry includes all bytes up to the next one; for example, 128 represents 1 byte to 128 bytes. About 90% of the messages are less than 512 bytes, which represents about 40% of the total bytes transferred. (b) Effective bandwidth versus message size measured on SDR and DDR InfiniBand networks running MVAPICH (<http://nowlab.cse.ohio-state.edu/projects/mpi-iba>) with OS bypass (native) and without (IPoIB).

Benchmark suite is also measured to have about 75% of its messages below 512 bytes (plots not shown). Many applications send far more small messages than large ones, particularly since requests and acknowledgments are more frequent than data responses and block writes.

InfiniBand reduces protocol processing overhead by allowing it to be offloaded from the host computer to a controller on the InfiniBand network interface card. The benefits of protocol offloading and bypassing the operating system are shown in Figure F.32(b) for MVAPICH, a widely used implementation of MPI over InfiniBand. Effective bandwidth is plotted against message size for MVAPICH configured in two modes and two network speeds. One mode runs IPoIB, in which InfiniBand communication is handled by the IP layer implemented by the host's operating system (i.e., no OS bypass). The other mode runs MVAPICH directly over VAPI, which is the native Mellanox InfiniBand interface that offloads transport protocol processing to the channel adapter hardware (i.e., OS bypass). Results are shown for 10 Gbps single data rate (SDR) and 20 Gbps double data rate (DDR) InfiniBand networks. The results clearly show that offloading the protocol processing and bypassing the OS significantly reduce sending and receiving overhead to allow near wire-speed effective bandwidth to be achieved.

Ethernet: The Local Area Network

Ethernet has been extraordinarily successful as a LAN—from the 10 Mbit/sec standard proposed in 1978 used practically everywhere today to the more recent 10 Gbit/sec standard that will likely be widely used. Many classes of computers include Ethernet as a standard communication interface. Ethernet, codified as IEEE standard 802.3, is a packet-switched network that routes packets using the destination address. It was originally designed for coaxial cable but today uses primarily Cat5E copper wire, with optical fiber reserved for longer distances and higher bandwidths. There is even a wireless version (802.11), which is testimony to its ubiquity.

Over a 20-year span, computers became thousands of times faster than they were in 1978, but the shared media Ethernet network remained the same. Hence, engineers had to invent temporary solutions until a faster, higher-bandwidth network became available. One solution was to use multiple Ethernets to interconnect machines and to connect those Ethernets with internetworking devices that could transfer traffic from one Ethernet to another, as needed. Such devices allow individual Ethernets to operate in parallel, thereby increasing the aggregate interconnection bandwidth of a collection of computers. In effect, these devices provide similar functionality to the switches described previously for point-to-point networks.

Figure F.33 shows the potential parallelism that can be gained. Depending on how they pass traffic and what kinds of interconnections they can join together, these devices have different names:

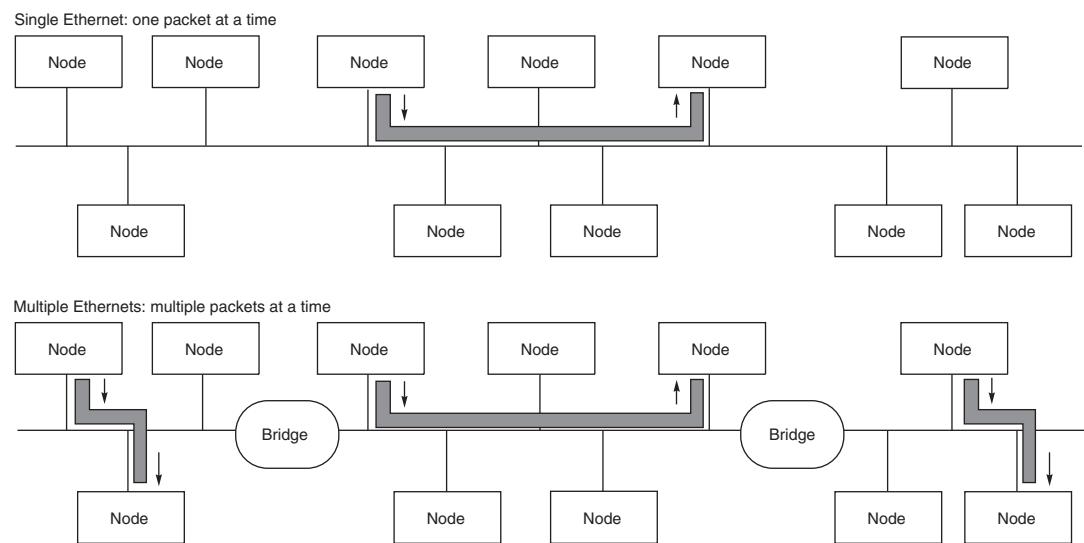


Figure F.33 The potential increased bandwidth of using many Ethernets and bridges.

- *Bridges*—These devices connect LANs together, passing traffic from one side to another depending on the addresses in the packet. Bridges operate at the Ethernet protocol level and are usually simpler and cheaper than routers, discussed next. Using the notation of the OSI model described in the next section (see Figure F.36 on page F-85), bridges operate at layer 2, the data link layer.
- Routers or *gateways*—These devices connect LANs to WANs, or WANs to WANs, and resolve incompatible addressing. Generally slower than bridges, they operate at OSI layer 3, the network layer. WAN routers divide the network into separate smaller subnets, which simplifies manageability and improves security.

The final internetworking devices are *hubs*, but they merely extend multiple segments into a single LAN. Thus, hubs do not help with performance, as only one message can transmit at a time. Hubs operate at OSI layer 1, called the physical

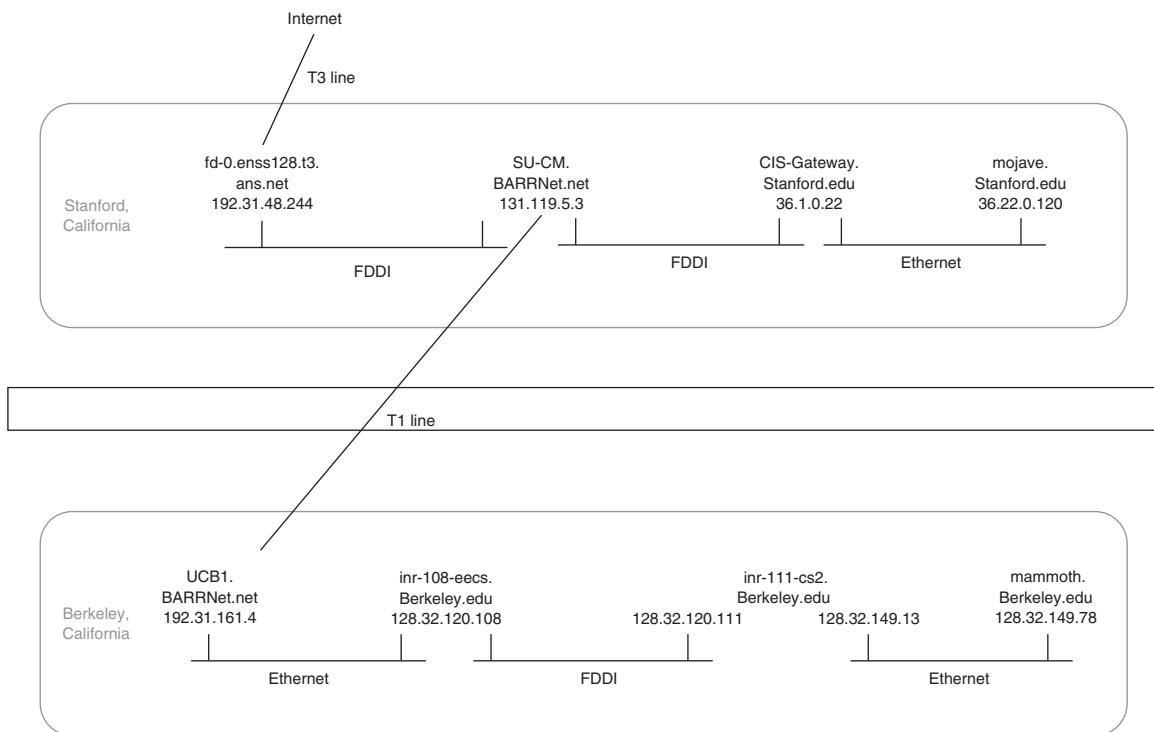


Figure F.34 The connection established between **mojave.stanford.edu** and **mammoth.berkeley.edu** (1995). FDDI is a 100 Mbit/sec LAN, while a T1 line is a 1.5 Mbit/sec telecommunications line and a T3 is a 45 Mbit/sec telecommunications line. BARRNet stands for Bay Area Research Network. Note that **inr-111-cs2.Berkeley.edu** is a router with two Internet addresses, one for each port.

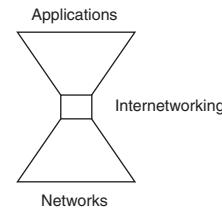


Figure F.35 The role of internetworking. The width indicates the relative number of items at each level.

Layer number	Layer name	Main function	Example protocol	Network component
7	Application	Used for applications specifically written to run over the network	FTP, DNS, NFS, http	Gateway, smart switch
6	Presentation	Translates from application to network format, and <i>vice versa</i>		Gateway
5	Session	Establishes, maintains, and ends sessions across the network	Named pipes, RPC	Gateway
4	Transport	Additional connection below the session layer	TCP	Gateway
3	Network	Translates logical network address and names to their physical address (e.g., computer name to MAC address)	IP	Router, ATM switch
2	Data Link	Turns packets into raw bits and at the receiving end turns bits into packets	Ethernet	Bridge, network interface card
1	Physical	Transmits raw bit stream over physical cable	IEEE 802	Hub

Figure F.36 The OSI model layers. Based on www.geocities.com/SiliconValley/Monitor/3131/ne/osimodel.html.

layer. Since these devices were not planned as part of the Ethernet standard, their ad hoc nature has added to the difficulty and cost of maintaining LANs.

As of 2011, Ethernet link speeds are available at 10, 100, 10,000, and 100,000 Mbits/sec. Although 10 and 100 Mbits/sec Ethernets share the media with multiple devices, 1000 Mbits/sec and above Ethernets rely on point-to-point links and switches. Ethernet switches normally use some form of store-and-forward.

Ethernet has no real flow control, dating back to its first instantiation. It originally used carrier sensing with exponential back-off (see page F-23) to arbitrate for the shared media. Some switches try to use that interface to retrofit their version of flow control, but flow control is not part of the Ethernet standard.

Wide Area Network: ATM

Asynchronous Transfer Mode (ATM) is a wide area networking standard set by the telecommunications industry. Although it flirted as competition to Ethernet as a LAN in the 1990s, ATM has since retreated to its WAN stronghold.

The telecommunications standard has scalable bandwidth built in. It starts at 155 Mbits/sec and scales by factors of 4 to 620 Mbits/sec, 2480 Mbits/sec, and so on. Since it is a WAN, ATM's medium is fiber, both single mode and multimode. Although it is a switched medium, unlike the other examples it relies on virtual connections for communication. ATM uses virtual channels for routing to multiplex different connections on a single network segment, thereby avoiding the inefficiencies of conventional connection-based networking. The WAN focus also led to store-and-forward switching. Unlike the other protocols, Figure F.30 shows ATM has a small, fixed-sized packet with 48 bytes of payload. It uses a credit-based flow control scheme as opposed to IP routers that do not implement flow control.

The reason for virtual connections and small packets is quality of service. Since the telecommunications industry is concerned about voice traffic, predictability matters as well as bandwidth. Establishing a virtual connection has less variability than connectionless networking, and it simplifies store-and-forward switching. The small, fixed packet also makes it simpler to have fast routers and switches. Toward that goal, ATM even offers its own protocol stack to compete with TCP/IP. Surprisingly, even though the switches are simple, the ATM suite of protocols is large and complex. The dream was a seamless infrastructure from LAN to WAN, avoiding the hodgepodge of routers common today. That dream has faded from inspiration to nostalgia.

F.9

Internetworking

Undoubtedly one of the most important innovations in the communications community has been internetworking. It allows computers on independent and incompatible networks to communicate reliably and efficiently. Figure F.34 illustrates the need to traverse between networks. It shows the networks and machines involved in transferring a file from Stanford University to the University of California at Berkeley, a distance of about 75 km.

The low cost of internetworking is remarkable. For example, it is vastly less expensive to send electronic mail than to make a coast-to-coast telephone call and leave a message on an answering machine. This dramatic cost improvement is achieved using the same long-haul communication lines as the telephone call, which makes the improvement even more impressive.

The enabling technologies for internetworking are software standards that allow reliable communication without demanding reliable networks. The underlying principle of these successful standards is that they were composed as a hierarchy of layers, each layer taking responsibility for a portion of the overall communication task. Each computer, network, and switch implements its layer of the standards, relying on the other components to faithfully fulfill their responsibilities. These layered software standards are called protocol families or protocol suites. They enable applications to work with any interconnection without extra work by the application programmer. Figure F.35 suggests the hierarchical model of communication.

The most popular internetworking standard is TCP/IP (Transmission Control Protocol/Internet Protocol). This protocol family is the basis of the humbly named Internet, which connects hundreds of millions of computers around the world. This popularity means TCP/IP is used even when communicating locally across compatible networks; for example, the network file system (NFS) uses IP even though it is very likely to be communicating across a homogenous LAN such as Ethernet. We use TCP/IP as our protocol family example; other protocol families follow similar lines. Section F.13 gives the history of TCP/IP.

The goal of a family of protocols is to simplify the standard by dividing responsibilities hierarchically among layers, with each layer offering services needed by the layer above. The application program is at the top, and at the bottom is the physical communication medium, which sends the bits. Just as abstract data types simplify the programmer's task by shielding the programmer from details of the implementation of the data type, this layered strategy makes the standard easier to understand.

There were many efforts at network protocols, which led to confusion in terms. Hence, Open Systems Interconnect (OSI) developed a model that popularized describing networks as a series of layers. Figure F.36 shows the model. Although all protocols do not exactly follow this layering, the nomenclature for the different layers is widely used. Thus, you can hear discussions about a simple layer 3 switch versus a layer 7 smart switch.

The key to protocol families is that communication occurs logically at the same level of the protocol in both sender and receiver, but services of the lower level implement it. This style of communication is called *peer-to-peer*. As an analogy, imagine that General A needs to send a message to General B on the battlefield. General A writes the message, puts it in an envelope addressed to General B, and gives it to a colonel with orders to deliver it. This colonel puts it in an envelope, and writes the name of the corresponding colonel who reports to General B, and gives it to a major with instructions for delivery. The major does the same thing and gives it to a captain, who gives it to a lieutenant, who gives it to a sergeant. The sergeant takes the envelope from the lieutenant, puts it into an envelope with the name of a sergeant who is in General B's division, and finds a private with orders to take the large envelope. The private borrows a motorcycle and delivers the envelope to the other sergeant. Once it arrives, it is passed up the chain of command, with each person removing an outer envelope with his name on it and passing on the inner envelope to his superior. As far as General B can tell, the note is from another general. Neither general knows who was involved in transmitting the envelope, nor how it was transported from one division to the other.

Protocol families follow this analogy more closely than you might think, as Figure F.37 shows. The original message includes a header and possibly a trailer sent by the lower-level protocol. The next-lower protocol in turn adds its own header to the message, possibly breaking it up into smaller messages if it is too large for this layer. Reusing our analogy, a long message from the general is divided and placed in several envelopes if it could not fit in one. This division of the message and appending of headers and trailers continues until the message

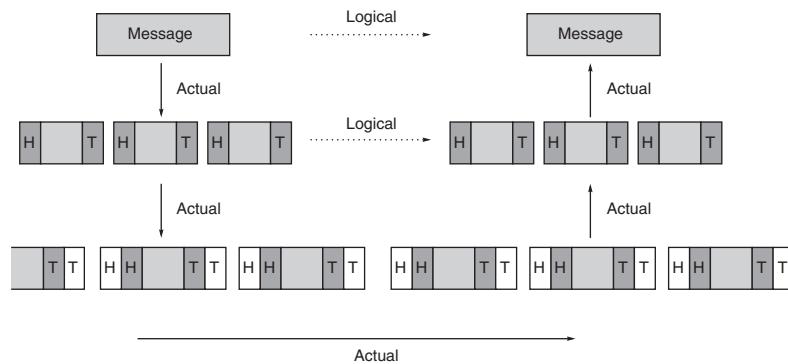


Figure F.37 A generic protocol stack with two layers. Note that communication is peer-to-peer, with headers and trailers for the peer added at each sending layer and removed by each receiving layer. Each layer offers services to the one above to shield it from unnecessary details.

descends to the physical transmission medium. The message is then sent to the destination. Each level of the protocol family on the receiving end will check the message at its level and peel off its headers and trailers, passing it on to the next higher level and putting the pieces back together. This nesting of protocol layers for a specific message is called a *protocol stack*, reflecting the last in, first out nature of the addition and removal of headers and trailers.

As in our analogy, the danger in this layered approach is the considerable latency added to message delivery. Clearly, one way to reduce latency is to reduce the number of layers, but keep in mind that protocol families define a standard but do not force how to implement the standard. Just as there are many ways to implement an instruction set architecture, there are many ways to implement a protocol family.

Our protocol stack example is TCP/IP. Let's assume that the bottom protocol layer is Ethernet. The next level up is the Internet Protocol or IP layer; the official term for an IP packet is a datagram. The IP layer routes the datagram to the destination machine, which may involve many intermediate machines or switches. IP makes a best effort to deliver the packets but does not guarantee delivery, content, or order of datagrams. The TCP layer above IP makes the guarantee of reliable, in-order delivery and prevents corruption of datagrams.

Following the example in Figure F.37, assume an application program wants to send a message to a machine via an Ethernet. It starts with TCP. The largest number of bytes that can be sent at once is 64 KB. Since the data may be much larger than 64 KB, TCP must divide them into smaller segments and reassemble them in proper order upon arrival. TCP adds a 20-byte header (Figure F.38) to every datagram and passes them down to IP. The IP layer above the physical layer adds a 20-byte header, also shown in Figure F.38. The data sent down from the IP level

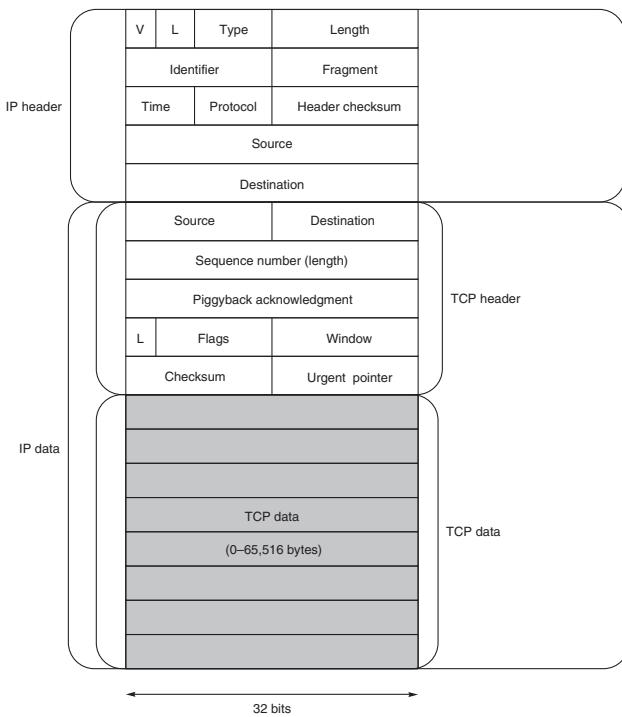


Figure F.38 The headers for IP and TCP. This drawing is 32 bits wide. The standard headers for both are 20 bytes, but both allow the headers to optionally lengthen for rarely transmitted information. Both headers have a length of header field (L) to accommodate the optional fields, as well as source and destination fields. The length field of the whole datagram is in a separate length field in IP, while TCP combines the length of the datagram with the sequence number of the datagram by giving the sequence number in bytes. TCP uses the checksum field to be sure that the datagram is not corrupted, and the sequence number field to be sure the datagrams are assembled into the proper order when they arrive. IP provides checksum error detection only for the header, since TCP has protected the rest of the packet. One optimization is that TCP can send a sequence of datagrams before waiting for permission to send more. The number of datagrams that can be sent without waiting for approval is called the *window*, and the window field tells how many bytes may be sent beyond the byte being acknowledged by this datagram. TCP will adjust the size of the window depending on the success of the IP layer in sending datagrams; the more reliable and faster it is, the larger TCP makes the window. Since the window slides forward as the data arrive and are acknowledged, this technique is called a *sliding window protocol*. The piggyback acknowledgment field of TCP is another optimization. Since some applications send data back and forth over the same connection, it seems wasteful to send a datagram containing only an acknowledgment. This piggyback field allows a datagram carrying data to also carry the acknowledgment for a previous transmission, “piggybacking” on top of a data transmission. The urgent pointer field of TCP gives the address within the datagram of an important byte, such as a break character. This pointer allows the application software to skip over data so that the user doesn’t have to wait for all prior data to be processed before seeing a character that tells the software to stop. The identifier field and fragment field of IP allow intermediary machines to break the original datagram into many smaller datagrams. A unique identifier is associated with the original datagram and placed in every fragment, with the fragment field saying which piece is which. The time-to-live field allows a datagram to be killed off after going through a maximum number of intermediate switches no matter where it is in the network. Knowing the maximum number of hops that it will take for a datagram to arrive—if it ever arrives—simplifies the protocol software. The protocol field identifies which possible upper layer protocol sent the IP datagram; in our case, it is TCP. The V (for version) and type fields allow different versions of the IP protocol software for the network. Explicit version numbering is included so that software can be upgraded gracefully machine by machine, without shutting down the entire network. Nowadays, version six of the Internet protocol (IPv6) was widely used.

to the Ethernet are sent in packets with the format shown in Figure F.30. Note that the TCP packet appears inside the data portion of the IP datagram, just as Figure F.37 suggests.

F.10

Crosscutting Issues for Interconnection Networks

This section describes five topics discussed in other chapters that are fundamentally impacted by interconnection networks, and *vice versa*.

Density-Optimized Processors versus SPEC-Optimized Processors

Given that people all over the world are accessing Web sites, it doesn't really matter where servers are located. Hence, many servers are kept at *collocation sites*, which charge by network bandwidth reserved and used and by space occupied and power consumed. Desktop microprocessors in the past have been designed to be as fast as possible at whatever heat could be dissipated, with little regard for the size of the package and surrounding chips. In fact, some desktop microprocessors from Intel and AMD as recently as 2006 burned as much as 130 watts! Floor space efficiency was also largely ignored. As a result of these priorities, power is a major cost for collocation sites, and processor density is limited by the power consumed and dissipated, including within the interconnect!

With the proliferation of portable computers (notebook sales exceeded desktop sales for the first time in 2005) and their reduced power consumption and cooling demands, the opportunity exists for using this technology to create considerably denser computation. For instance, the power consumption for the Intel Pentium M in 2006 was 25 watts, yet it delivered performance close to that of a desktop microprocessor for a wide set of applications. It is therefore conceivable that performance per watt or performance per cubic foot could replace performance per microprocessor as the important figure of merit. The key is that many applications already make use of large clusters, so it is possible that replacing 64 power-hungry processors with, say, 256 power-efficient processors could be cheaper yet be software compatible. This places greater importance on power- and performance-efficient interconnection network design.

The Google cluster is a prime example of this migration to many “cooler” processors versus fewer “hotter” processors. It uses racks of up to 80 Intel Pentium III 1 GHz processors instead of more power-hungry high-end processors. Other examples include blade servers consisting of 1-inch-wide by 7-inch-high rack unit blades designed based on mobile processors. The HP ProLiant BL10e G2 blade server supports up to 20 1-GHz ultra-low-voltage Intel Pentium M processors with a 400-MHz front-side bus, 1-MB L2 cache, and up to 1 GB memory. The Fujitsu Primergy BX300 blade server supports up to 20 1.4- or 1.6-GHz Intel Pentium M processors, each with 512 MB of memory expandable to 4 GB.

Smart Switches versus Smart Interface Cards

Figure F.39 shows a trade-off as to where intelligence can be located within a network. Generally, the question is whether to have either smarter network interfaces or smarter switches. Making one smarter generally makes the other simpler and less expensive. By having an inexpensive interface, it was possible for Ethernet to become standard as part of most desktop and server computers. Lower-cost switches were made available for people with small configurations, not needing sophisticated forwarding tables and spanning-tree protocols of larger Ethernet switches.

Myrinet followed the opposite approach. Its switches are dumb components that, other than implementing flow control and arbitration, simply extract the first byte from the packet header and use it to directly select the output port. No routing tables are implemented, so the intelligence is in the network interface cards (NICs). The NICs are responsible for providing support for efficient communication and for implementing a distributed protocol for network (re)configuration. InfiniBand takes a hybrid approach by offering lower-cost, less sophisticated interface cards called target channel adapters (or TCAs) for less demanding devices such as disks—in the hope that it can be included within some I/O devices—and by offering more expensive, powerful interface cards for hosts called host channel adapters (or HCAs). The switches implement routing tables.

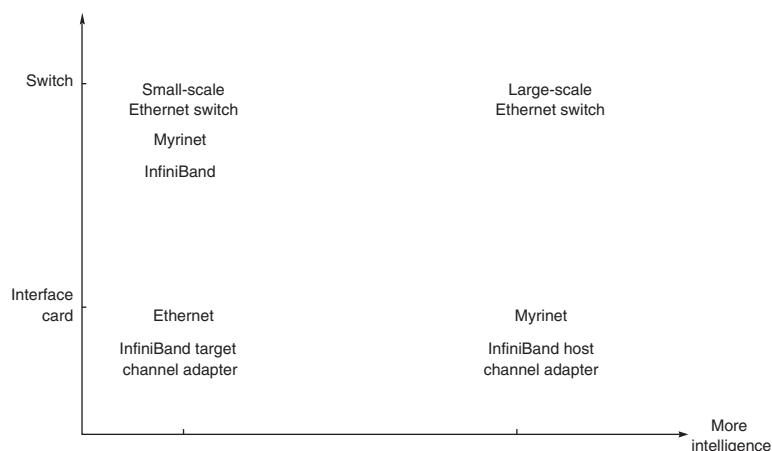


Figure F.39 Intelligence in a network: switch versus network interface card. Note that Ethernet switches come in two styles, depending on the size of the network, and that InfiniBand network interfaces come in two styles, depending on whether they are attached to a computer or to a storage device. Myrinet is a proprietary system area network.

Protection and User Access to the Network

A challenge is to ensure safe communication across a network without invoking the operating system in the common case. The Cray Research T3D supercomputer offers an interesting case study. Like the more recent Cray X1E, the T3D supports a global address space, so loads and stores can access memory across the network. Protection is ensured because each access is checked by the TLB. To support transfer of larger objects, a block transfer engine (BLT) was added to the hardware. Protection of access requires invoking the operating system before using the BLT to check the range of accesses to be sure there will be no protection violations.

Figure F.40 compares the bandwidth delivered as the size of the object varies for reads and writes. For very large reads (e.g., 512 KB), the BLT achieves the highest performance: 140 MB/sec. But simple loads get higher performance for 8 KB or less. For the write case, both achieve a peak of 90 MB/sec, presumably because of the limitations of the memory bus. But, for writes, the BLT can only match the performance of simple stores for transfers of 2 MB; anything smaller and it's faster to send stores. Clearly, a BLT that can avoid invoking the operating system in the common case would be more useful.

Efficient Interface to the Memory Hierarchy versus the Network

Traditional evaluations of processor performance, such as SPECint and SPECfp, encourage integration of the memory hierarchy with the processor as the efficiency of the memory hierarchy translates directly into processor performance. Hence,

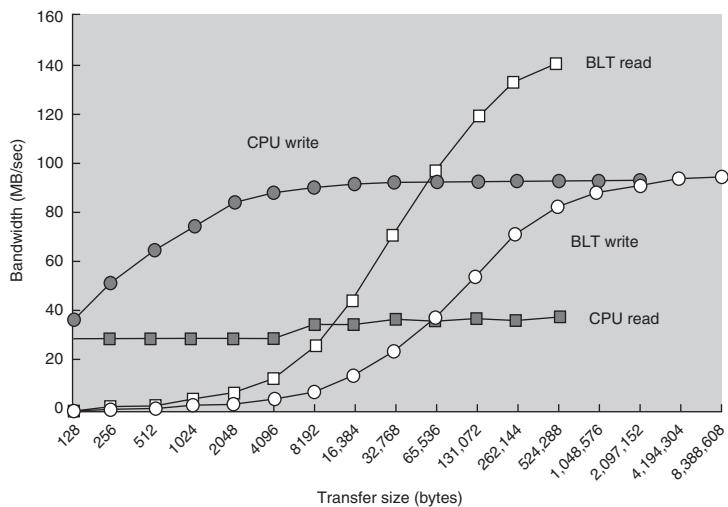


Figure F.40 Bandwidth versus transfer size for simple memory access instructions versus a block transfer device on the Cray Research T3D. (From Arpaci et al. [1995].)

microprocessors have multiple levels of caches on chip along with buffers for writes. Because benchmarks such as SPECint and SPECfp do not reward good interfaces to interconnection networks, many machines make the access time to the network delayed by the full memory hierarchy. Writes must lumber their way through full write buffers, and reads must go through the cycles of first-, second-, and often third-level cache misses before reaching the interconnection network. This hierarchy results in newer systems having higher latencies to the interconnect than older machines.

Let's compare three machines from the past: a 40-MHz SPARCstation-2, a 50-MHz SPARCstation-20 without an external cache, and a 50-MHz SPARCstation-20 with an external cache. According to SPECint95, this list is in order of increasing performance. The time to access the I/O bus (S-bus), however, increases in this sequence: 200 ns, 500 ns, and 1000 ns. The SPARCstation-2 is fastest because it has a single bus for memory and I/O, and there is only one level to the cache. The SPARCstation-20 memory access must first go over the memory bus (M-bus) and then to the I/O bus, adding 300 ns. Machines with a second-level cache pay an extra penalty of 500 ns before accessing the I/O bus.

Compute-Optimized Processors versus Receiver Overhead

The overhead to receive a message likely involves an interrupt, which bears the cost of flushing and then restarting the processor pipeline, if not offloaded. As mentioned earlier, reading network status and receiving data from the network interface likely operate at cache miss speeds. If microprocessors become more superscalar and go to even faster clock rates, the number of missed instruction issue opportunities per message reception will likely rise to unacceptable levels.

F.11

Fallacies and Pitfalls

Myths and hazards are widespread with interconnection networks. This section mentions several warnings, so proceed carefully.

Fallacy

The interconnection network is very fast and does not need to be improved

The interconnection network provides certain functionality to the system, very much like the memory and I/O subsystems. It should be designed to allow processors to execute instructions at the maximum rate. The interconnection network subsystem should provide high enough bandwidth to keep from continuously entering saturation and becoming an overall system bottleneck.

In the 1980s, when wormhole switching was introduced, it became feasible to design large-diameter topologies with single-chip switches so that the bandwidth capacity of the network was not the limiting factor. This led to the flawed belief that interconnection networks need no further improvement.

Since the 1980s, much attention has been placed on improving processor performance, but comparatively less has been focused on interconnection networks. As technology advances, the interconnection network tends to represent an increasing fraction of system resources, cost, power consumption, and various other attributes that impact functionality and performance. Scaling the bandwidth simply by overdimensioning certain network parameters is no longer a cost-viable option. Designers must carefully consider the end-to-end interconnection network design in concert with the processor, memory, and I/O subsystems in order to achieve the required cost, power, functionality, and performance objectives of the entire system. An obvious case in point is multicore processors with on-chip networks.

Fallacy *Bisection bandwidth is an accurate cost constraint of a network*

Despite being very popular, bisection bandwidth has never been a practical constraint on the implementation of an interconnection network, although it may be one in future designs. It is more useful as a performance measure than as a cost measure. Chip pin-outs are the more realistic bandwidth constraint.

Pitfall *Using bandwidth (in particular, bisection bandwidth) as the only measure of network performance*

It seldom is the case that aggregate network bandwidth (likewise, network bisection bandwidth) is the end-to-end bottlenecking point across the network. Even if it were the case, networks are almost never 100% efficient in transporting packets across the bisection (i.e., $\rho < 100\%$) nor at receiving them at network endpoints (i.e., $\sigma < 100\%$). The former is highly dependent upon routing, switching, arbitration, and other such factors while both the former and the latter are highly dependent upon traffic characteristics. Ignoring these important factors and concentrating only on raw bandwidth can give very misleading performance predictions. For example, it is perfectly conceivable that a network could have higher aggregate bandwidth and/or bisection bandwidth relative to another network but also have lower measured performance!

Apparently, given sophisticated protocols like TCP/IP that maximize delivered bandwidth, many network companies believe that there is only one figure of merit for networks. This may be true for some applications, such as video streaming, where there is little interaction between the sender and the receiver. Many applications, however, are of a request-response nature, and so for every large message there must be one or more small messages. One example is NFS.

Figure F.41 compares a shared 10-Mbit/sec Ethernet LAN to a switched 155-Mbit/sec ATM LAN for NFS traffic. Ethernet drivers were better tuned than the ATM drivers, such that 10-Mbit/sec Ethernet was faster than 155-Mbit/sec ATM for payloads of 512 bytes or less. Figure F.41 shows the overhead time, transmission time, and total time to send all the NFS messages over Ethernet and ATM. The peak link speed of ATM is 15 times faster, and the measured link speed for 8-KB messages is almost 9 times faster. Yet, the higher overheads offset the benefits so that ATM would transmit NFS traffic only 1.2 times faster.

Size	Number of messages	Overhead (sec)		Number of data bytes	Transmission (sec)		Total time (sec)	
		ATM	Ethernet		ATM	Ethernet	ATM	Ethernet
32	771,060	532	389	33,817,052	4	48	536	436
64	56,923	39	29	4,101,088	0	5	40	34
96	4,082,014	2817	2057	428,346,316	46	475	2863	2532
128	5,574,092	3846	2809	779,600,736	83	822	3929	3631
160	328,439	227	166	54,860,484	6	56	232	222
192	16,313	11	8	3,316,416	0	3	12	12
224	4820	3	2	1,135,380	0	1	3	4
256	24,766	17	12	9,150,720	1	9	18	21
512	32,159	22	16	25,494,920	3	23	25	40
1024	69,834	48	35	70,578,564	8	72	56	108
1536	8842	6	4	15,762,180	2	14	8	19
2048	9170	6	5	20,621,760	2	19	8	23
2560	20,206	14	10	56,319,740	6	51	20	61
3072	13,549	9	7	43,184,992	4	39	14	46
3584	4200	3	2	16,152,228	2	14	5	17
4096	67,808	47	34	285,606,596	29	255	76	290
5120	6143	4	3	35,434,680	4	32	8	35
6144	5858	4	3	37,934,684	4	34	8	37
7168	4140	3	2	31,769,300	3	28	6	30
8192	287,577	198	145	2,390,688,480	245	2132	444	2277
Total	11,387,913	7858	5740	4,352,876,316	452	4132	8310	9872

Figure F.41 Total time on a 10-Mbit Ethernet and a 155-Mbit ATM, calculating the total overhead and transmission time separately. Note that the size of the headers needs to be added to the data bytes to calculate transmission time. The higher overhead of the software driver for ATM offsets the higher bandwidth of the network. These measurements were performed in 1994 using SPARCstation 10s, the ForeSystems SBA-200 ATM interface card, and the Fore Systems ASX-200 switch. (NFS measurements taken by Mike Dahlin of the University of California–Berkeley.)

Pitfall *Not providing sufficient reception link bandwidth, which causes the network end nodes to become even more of a bottleneck to performance*

Unless the traffic pattern is a permutation, several packets will concurrently arrive at some destinations when most source devices inject traffic, thus producing contention. If this problem is not addressed, contention may turn into congestion that will spread across the network. This can be dealt with by analyzing traffic patterns and providing extra reception bandwidth. For example, it is possible to implement more reception bandwidth than injection bandwidth. The IBM Blue Gene/L, for example, implements an on-chip switch with 7-bit

injection and 12-bit reception links, where the reception BW equals the aggregate switch input link BW.

Pitfall *Using high-performance network interface cards but forgetting about the I/O subsystem that sits between the network interface and the host processor*

This issue is related to the previous one. Messages are usually composed in user space buffers and later sent by calling a send function from the communications library. Alternatively, a cache controller implementing a cache coherence protocol may compose a message in some SANs and in OCNs. In both cases, messages have to be copied to the network interface memory before transmission. If the I/O bandwidth is lower than the link bandwidth or introduces significant overhead, this is going to affect communication performance significantly. As an example, the first 10-Gigabit Ethernet cards in the market had a PCI-X bus interface for the system with a significantly lower bandwidth than 10 Gbps.

Fallacy *Zero-copy protocols do not require copying messages or fragments from one buffer to another*

Traditional communication protocols for computer networks allow access to communication devices only through system calls in supervisor mode. As a consequence of this, communication routines need to copy the corresponding message from the user buffer to a kernel buffer when sending a message. Note that the communication protocol may need to keep a copy of the message for retransmission in case of error, and the application may modify the contents of the user buffer once the system call returns control to the application. This buffer-to-buffer copy is eliminated in zero-copy protocols because the communication routines are executed in user space and protocols are much simpler.

However, messages still need to be copied from the application buffer to the memory in the network interface card (NIC) so that the card hardware can transmit it from there through to the network. Although it is feasible to eliminate this copy by allocating application message buffers directly in the NIC memory (and, indeed, this is done in some protocols), this may not be convenient in current systems because access to the NIC memory is usually performed through the I/O subsystem, which usually is much slower than accessing main memory. Thus, it is generally more efficient to compose the message in main memory and let DMA devices take care of the transfer to the NIC memory.

Moreover, what few people count is the copy from where the message fragments are computed (usually the ALU, with results stored in some processor register) to main memory. Some systolic-like architectures in the 1980s, like the iWarp, were able to directly transmit message fragments from the processor to the network, effectively eliminating all the message copies. This is the approach taken in the Cray X1E shared-memory multiprocessor supercomputer.

Similar comments can be made regarding the reception side; however, this does not mean that zero-copy protocols are inefficient. These protocols represent the most efficient kind of implementation used in current systems.

Pitfall *Ignoring software overhead when determining performance*

Low software overhead requires cooperation with the operating system as well as with the communication libraries, but even with protocol offloading it continues to dominate the hardware overhead and must not be ignored. Figures F.32 and F.41 give two examples, one for a SAN standard and the other for a WAN standard. Other examples come from proprietary SANs for supercomputers. The Connection Machine CM-5 supercomputer in the early 1990s had a software overhead of 20 μ s to send a message and a hardware overhead of only 0.5 μ s. The first Intel Paragon supercomputer built in the early 1990s had a hardware overhead of just 0.2 μ s, but the initial release of the software had an overhead of 250 μ s. Later releases reduced this overhead down to 25 μ s and, more recently, down to only a few microseconds, but this still dominates the hardware overhead. The IBM Blue Gene/L has an MPI sending/receiving overhead of approximately 3 μ s, only a third of which (at most) is attributed to the hardware.

This pitfall is simply Amdahl's law applied to networks: Faster network hardware is superfluous if there is not a corresponding decrease in software overhead. The software overhead is much reduced these days with OS bypass, lightweight protocols, and protocol offloading down to a few microseconds or less, typically, but it remains a significant factor in determining performance.

Fallacy *MINs are more cost-effective than direct networks*

A MIN is usually implemented using significantly fewer switches than the number of devices that need to be connected. On the other hand, direct networks usually include a switch as an integral part of each node, thus requiring as many switches as nodes to interconnect. However, nothing prevents the implementation of nodes with multiple computing devices on it (e.g., a multicore processor with an on-chip switch) or with several devices attached to each switch (i.e., bristling). In these cases, a direct network may be as (or even more) cost-effective as a MIN. Note that, for a MIN, several network interfaces may be required at each node to match the bandwidth delivered by the multiple links per node provided by the direct network.

Fallacy *Low-dimensional direct networks achieve higher performance than high-dimensional networks such as hypercubes*

This conclusion was drawn by several studies that analyzed the optimal number of dimensions under the main physical constraint of bisection bandwidth. However, most of those studies did not consider link pipelining, considered only very short links, and/or did not consider switch architecture design constraints. The misplaced assumption that bisection bandwidth serves as the main limit did not help matters. Nowadays, most researchers and designers believe that high-radix switches are more cost-effective than low-radix switches, including some who concluded the opposite before.

Fallacy *Wormhole switching achieves better performance than other switching techniques*

Wormhole switching delivers the same no-load latency as other pipelined switching techniques, like virtual cut-through switching. The introduction of wormhole switches in the late 1980s coinciding with a dramatic increase in network bandwidth led many to believe that wormhole switching was the main reason for the performance boost. Instead, most of the performance increase came from a drastic increase in link bandwidth, which, in turn, was enabled by the ability of wormhole switching to buffer packet fragments using on-chip buffers, instead of using the node's main memory or some other off-chip source for that task. More recently, much larger on-chip buffers have become feasible, and virtual cutthrough achieved the same no-load latency as wormhole while delivering much higher throughput. This did not mean that wormhole switching was dead. It continues to be the switching technique of choice for applications in which only small buffers should be used (e.g., perhaps for on-chip networks).

Fallacy *Implementing a few virtual channels always increases throughput by allowing packets to pass through blocked packets ahead*

In general, implementing a few virtual channels in a wormhole switch is a good idea because packets are likely to pass blocked packets ahead of them, thus reducing latency and significantly increasing throughput. However, the improvements are not as dramatic for virtual cut-through switches. In virtual cut-through, buffers should be large enough to store several packets. As a consequence, each virtual channel may introduce HOL blocking, possibly degrading performance at high loads. Adding virtual channels increases cost, but it may deliver little additional performance unless there are as many virtual channels as switch ports and packets are mapped to virtual channels according to their destination (i.e., virtual output queueing). It is certainly the case that virtual channels can be useful in virtual cut-through networks to segregate different traffic classes, which can be very beneficial. However, multiplexing the packets over a physical link on a flit-by-flit basis causes all the packets from different virtual channels to get delayed. The average packet delay is significantly shorter if multiplexing takes place on a packet-by-packet basis, but in this case packet size should be bounded to prevent any one packet from monopolizing the majority of link bandwidth.

Fallacy *Adaptive routing causes out-of-order packet delivery, thus introducing too much overhead needed to reorder packets at the destination device*

Adaptive routing allows packets to follow alternative paths through the network depending on network traffic; therefore, adaptive routing usually introduces out-of-order packet delivery. However, this does not necessarily imply that reordering packets at the destination device is going to introduce a large overhead, making adaptive routing not useful. For example, the most efficient adaptive routing algorithms to date support fully adaptive routing in some virtual channels but required

deterministic routing to be implemented in some other virtual channels in order to prevent deadlocks (à la the IBM Blue Gene/L). In this case, it is very easy to select between adaptive and deterministic routing for each individual packet. A single bit in the packet header can indicate to the switches whether all the virtual channels can be used or only those implementing deterministic routing. This hardware support can be used as indicated below to eliminate packet reordering overhead at the destination.

Most communication protocols for parallel computers and clusters implement two different protocols depending on message size. For short messages, an eager protocol is used in which messages are directly transmitted, and the receiving nodes use some preallocated buffer to temporarily store the incoming message. On the other hand, for long messages, a rendezvous protocol is used. In this case, a control message is sent first, requesting the destination node to allocate a buffer large enough to store the entire message. The destination node confirms buffer allocation by returning an acknowledgment, and the sender can proceed with fragmenting the message into bounded-size packets, transmitting them to the destination.

If eager messages use only deterministic routing, it is obvious that they do not introduce any reordering overhead at the destination. On the other hand, packets belonging to a long message can be transmitted using adaptive routing. As every packet contains the sequence number within the message (or the offset from the beginning of the message), the destination node can store every incoming packet directly in its correct location within the message buffer, thus incurring no overhead with respect to using deterministic routing. The only thing that differs is the completion condition. Instead of checking that the last packet in the message has arrived, it is now necessary to count the arrived packets, notifying the end of reception when the count equals the message size. Taking into account that long messages, even if not frequent, usually consume most of the network bandwidth, it is clear that most packets can benefit from adaptive routing without introducing reordering overhead when using the protocol described above.

Fallacy *Adaptive routing by itself always improves network fault tolerance because it allows packets to follow alternative paths*

Adaptive routing by itself is not enough to tolerate link and/or switch failures. Some mechanism is required to detect failures and notify them, so that the routing logic could exclude faulty paths and use the remaining ones. Moreover, while a given link or switch failure affects a certain number of paths when using deterministic routing, many more source/destination pairs could be affected by the same failure when using adaptive routing. As a consequence of this, some switches implementing adaptive routing transition to deterministic routing in the presence of failures. In this case, failures are usually tolerated by sending messages through alternative paths from the source node. As an example, the Cray T3E implements direction-order routing to tolerate a few failures. This fault-tolerant routing technique avoids cycles in the use of resources by crossing directions in order

(e.g., $X+$, Y_+ , Z_+ , Z_- , Y_- , then X_-). At the same time, it provides an easy way to send packets through nonminimal paths, if necessary, to avoid crossing faulty components. For instance, a packet can be initially forwarded a few hops in the X_+ direction even if it has to go in the X_- direction at some point later.

Pitfall *Trying to provide features only within the network versus end-to-end*

The concern is that of providing at a lower level the features that can only be accomplished at the highest level, thus only partially satisfying the communication demand. Saltzer, Reed, and Clark [1984] gave the end-to-end argument as follows:

The function in question can completely and correctly be specified only with the knowledge and help of the application standing at the endpoints of the communication system. Therefore, providing that questioned function as a feature of the communication system itself is not possible. [page 278]

Their example of the pitfall was a network at MIT that used several gateways, each of which added a checksum from one gateway to the next. The programmers of the application assumed that the checksum guaranteed accuracy, incorrectly believing that the message was protected while stored in the memory of each gateway. One gateway developed a transient failure that swapped one pair of bytes per million bytes transferred. Over time, the source code of one operating system was repeatedly passed through the gateway, thereby corrupting the code. The only solution was to correct infected source files by comparing them to paper listings and repairing code by hand! Had the checksums been calculated and checked by the application running on the end systems, safety would have been ensured.

There is a useful role for intermediate checks at the link level, however, provided that end-to-end checking is available. End-to-end checking may show that something is broken between two nodes, but it doesn't point to where the problem is. Intermediate checks can discover the broken component.

A second issue regards performance using intermediate checks. Although it is sufficient to retransmit the whole in case of failures from the end point, it can be much faster to retransmit a portion of the message at an intermediate point rather than wait for a time-out and a full message retransmit at the end point.

Pitfall *Relying on TCP/IP for all networks, regardless of latency, bandwidth, or software requirements*

The network designers on the first workstations decided it would be elegant to use a single protocol stack no matter where the destination of the message: Across a room or across an ocean, the TCP/IP overhead must be paid. This might have been a wise decision back then, especially given the unreliability of early Ethernet hardware, but it sets a high software overhead barrier for commercial systems of today. Such an obstacle lowers the enthusiasm for low-latency network interface hardware and low-latency interconnection networks if the software is just going to waste hundreds of microseconds when the message must travel only dozens of meters or less. It also can use significant processor resources. One rough rule of

thumb is that each Mbit/sec of TCP/IP bandwidth needs about 1 MHz of processor speed, so a 1000-Mbit/sec link could saturate a processor with an 800- to 1000-MHz clock.

The flip side is that, from a software perspective, TCP/IP is the most desirable target since it is the most connected and, hence, provides the largest number of opportunities. The downside of using software optimized to a particular LAN or SAN is that it is limited. For example, communication from a Java program depends on TCP/IP, so optimization for another protocol would require creation of glue software to interface Java to it.

TCP/IP advocates point out that the protocol itself is theoretically not as burdensome as current implementations, but progress has been modest in commercial systems. There are also TCP/IP offloading engines in the market, with the hope of preserving the universal software model while reducing processor utilization and message latency. If processors continue to improve much faster than network speeds, or if multiple processors become ubiquitous, software TCP/IP may become less significant for processor utilization and message latency.

F.12

Concluding Remarks

Interconnection network design is one of the most exciting areas of computer architecture development today. With the advent of new multicore processor paradigms and advances in traditional multiprocessor/cluster systems and the Internet, many challenges and opportunities exist for interconnect architecture innovation. These apply to all levels of computer systems: communication between cores on a chip, between chips on a board, between boards in a system, and between computers in a machine room, over a local area and across the globe. Irrespective of their domain of application, interconnection networks should transfer the maximum amount of information within the least amount of time for given cost and power constraints so as not to bottleneck the system. Topology, routing, arbitration, switching, and flow control are among some of the key concepts in realizing such high-performance designs.

The design of interconnection networks is end-to-end: It includes injection links, reception links, and the interfaces at network end points as much as it does the topology, switches, and links within the network fabric. It is often the case that the bandwidth and overhead at the end node interfaces are the bottleneck, yet many mistakenly think of the interconnection network to mean only the network fabric. This is as bad as processor designers thinking of computer architecture to mean only the instruction set architecture or only the microarchitecture! End-to-end issues and understanding of the traffic characteristics make the design of interconnection networks challenging and very much relevant even today. For instance, the need for low end-to-end latency is driving the development of efficient network interfaces located closer to the processor/memory controller. We may soon see most multicore processors used in multiprocessor systems implementing network interfaces on-chip,

devoting some core(s) to execute communication tasks. This is already the case for the IBM Blue Gene/L supercomputer, which uses one of its two cores on each processor chip for this purpose.

Networking has a long way to go from its humble shared-media beginnings. It is in “catch-up” mode, with switched-media point-to-point networks only recently displacing traditional bus-based networks in many networking domains, including on chip, I/O, and the local area. We are not near any performance plateaus, so we expect rapid advancement of WANs, LANs, SANs, and especially OCNs in the near future. Greater interconnection network performance is key to the information- and communication-centric vision of the future of our field, which, so far, has benefited many millions of people around the world in various ways. As the quotes at the beginning of this appendix suggest, this revolution in *two-way* communication is at the heart of changes in the form of our human associations and actions.

Acknowledgments

We express our sincere thanks to the following persons who, in some way, have contributed to the contents of the previous edition of the appendix: Lei Chai, Scott Clark, José Flích, Jose Manuel García, Paco Gilabert, Rama Govindaraju, Manish Gupta, Wai Hong Ho, Siao Jer, Steven Keckler, Dhabaleswar (D.K.) Panda, Fabrizio Petruini, Steve Scott, Jeonghee Shin, Craig Stunkel, Sayantan Sur, Michael B. Taylor, and Bilal Zafar. We especially appreciate the new contributions of Jose Flích to this edition of the appendix.

F.13

Historical Perspective and References

This appendix has taken the perspective that interconnection networks for very different domains—from on-chip networks within a processor chip to wide area networks connecting computers across the globe—share many of the same concerns. With this, interconnection network concepts are presented in a unified way, irrespective of their application; however, their histories are vastly different, as evidenced by the different solutions adopted to address similar problems. The lack of significant interaction between research communities from the different domains certainly contributed to the diversity of implemented solutions. Highlighted below are relevant readings on each topic. In addition, good general texts featuring WAN and LAN networking have been written by Davie, Peterson, and Clark [1999] and by Kurose and Ross [2001]. Good texts focused on SANs for multiprocessors and clusters have been written by Duato, Yalamanchili, and Ni [2003] and by Dally and Towles [2004]. An informative chapter devoted to dead-lock resolution in interconnection networks was written by Pinkston [2004]. Finally, an edited work by Jantsch and Tenhunen [2003] on OCNs for multicore processors and system-on-chips is also interesting reading.

Wide Area Networks

Wide area networks are the earliest of the data interconnection networks. The forerunner of the Internet is the ARPANET, which in 1969 connected computer science departments across the United States that had research grants funded by the Advanced Research Project Agency (ARPA), a U.S. government agency. It was originally envisioned as using reliable communications at lower levels. Practical experience with failures of the underlying technology led to the failure-tolerant TCP/IP, which is the basis for the Internet today. Vint Cerf and Robert Kahn are credited with developing the TCP/IP protocols in the mid-1970s, winning the ACM Software Award in recognition of that achievement. Kahn [1972] is an early reference on the ideas of ARPANET. For those interested in learning more about TCP/IP, Stevens [1994–1996] has written classic books on the topic.

In 1975, there were roughly 100 networks in the ARPANET; in 1983, only 200. In 1995, the Internet encompassed 50,000 networks worldwide, about half of which were in the United States. That number is hard to calculate now, but the number of IP hosts grew by a factor of 15 from 1995 to 2000, reaching 100 million Internet hosts by the end of 2000. It has grown much faster since then. With most service providers assigning dynamic IP addresses, many local area networks using private IP addresses, and with most networks allowing wireless connections, the total number of hosts in the Internet is nearly impossible to compute. In July 2005, the Internet Systems Consortium (www.isc.org) estimated more than 350 million Internet hosts, with an annual increase of about 25% projected. Although key government networks made the Internet possible (i.e., ARPANET and NSFNET), these networks have been taken over by the commercial sector, allowing the Internet to thrive. But major innovations to the Internet are still likely to come from government-sponsored research projects rather than from the commercial sector. The National Science Foundation's Global Environment for Network Innovation (GENI) initiative is an example of this.

The most exciting application of the Internet is the World Wide Web, developed in 1989 by Tim Berners-Lee, a programmer at the European Center for Particle Research (CERN), for information access. In 1992, a young programmer at the University of Illinois, Marc Andreessen, developed a graphical interface for the Web called Mosaic. It became immensely popular. He later became a founder of Netscape, which popularized commercial browsers. In May 1995, at the time of the second edition of this book, there were over 30,000 Web pages, and the number was doubling every two months. During the writing of the third edition of this text, there were more than 1.3 billion Web pages. In December 2005, the number of Web servers approached 75 million, having increased by 30% during that same year.

Asynchronous Transfer Mode (ATM) was an attempt to design the definitive communication standard. It provided good support for data transmission as well as digital voice transmission (i.e., phone calls). From a technical point of view, it combined the best from packet switching and circuit switching, also providing excellent support for providing quality of service (QoS). Alles [1995] offers a good

survey on ATM. In 1995, no one doubted that ATM was going to be the future for this community. Ten years later, the high equipment and personnel training costs basically killed ATM, and we returned back to the simplicity of TCP/IP. Another important blow to ATM was its defeat by the Ethernet family in the LAN domain, where packet switching achieved significantly lower latencies than ATM, which required establishing a connection before data transmission. ATM connectionless servers were later introduced in an attempt to fix this problem, but they were expensive and represented a central bottleneck in the LAN.

Finally, WANs today rely on optical fiber. Fiber technology has made so many advances that today WAN fiber bandwidth is often underutilized. The main reason for this is the commercial introduction of wavelength division multiplexing (WDM), which allows each fiber to transmit many data streams simultaneously over different wavelengths, thus allowing three orders of magnitude bandwidth increase in just one generation, that is, 3 to 5 years (a good text by Senior [1993] discusses optical fiber communications). However, IP routers may still become a bottleneck. At 10- to 40-Gbps link rates, and with thousands of ports in large core IP routers, packets must be processed very quickly—that is, within a few tens of nanoseconds. The most time-consuming operation is routing. The way IP addresses have been defined and assigned to Internet hosts makes routing very complicated, usually requiring a complex search in a tree structure for every packet. Network processors have become popular as a cost-effective solution for implementing routing and other packet-filtering operations. They usually are RISC-like and highly multi-threaded and implement local stores instead of caches.

Local Area Networks

ARPA's success with wide area networks led directly to the most popular local area networks. Many researchers at Xerox Palo Alto Research Center had been funded by ARPA while working at universities, so they all knew the value of networking. In 1974, this group invented the Alto, the forerunner of today's desktop computers [Thacker et al. 1982], and the Ethernet [Metcalfe and Boggs 1976], today's LAN. This group—David Boggs, Butler Lampson, Ed McCreight, Bob Sproul, and Chuck Thacker—became luminaries in computer science and engineering, collecting a treasure chest of awards among them.

This first Ethernet provided a 3-Mbit/sec interconnection, which seemed like an unlimited amount of communication bandwidth with computers of that era. It relied on the interconnect technology developed for the cable television industry. Special microcode support gave a round-trip time of 50 μ s for the Alto over Ethernet, which is still a respectable latency. It was Boggs' experience as a ham radio operator that led to a design that did not need a central arbiter, but instead listened before use and then varied back-off times in case of conflicts.

The announcement by Digital Equipment Corporation, Intel, and Xerox of a standard for 10-Mbit/sec Ethernet was critical to the commercial success of

Ethernet. This announcement short-circuited a lengthy IEEE standards effort, which eventually did publish IEEE 802.3 as a standard for Ethernet.

There have been several unsuccessful candidates that have tried to replace the Ethernet. The Fiber Data Distribution Interconnect (FDDI) committee, unfortunately, took a very long time to agree on the standard, and the resulting interfaces were expensive. It was also a shared medium when switches were becoming affordable. ATM also missed the opportunity in part because of the long time to standardize the LAN version of ATM, and in part because of the high latency and poor behavior of ATM connectionless servers, as mentioned above. InfiniBand for the reasons discussed below has also faltered. As a result, Ethernet continues to be the absolute leader in the LAN environment, and it remains a strong opponent in the high-performance computing market as well, competing against the SANs by delivering high bandwidth at low cost. The main drawback of Ethernet for high-end systems is its relatively high latency and lack of support in most interface cards to implement the necessary protocols.

Because of failures of the past, LAN modernization efforts have been centered on extending Ethernet to lower-cost media such as unshielded twisted pair (UTP), switched interconnects, and higher link speeds as well as to new domains such as wireless communication. Practically all new PC motherboards and laptops implement a Fast/Gigabit Ethernet port (100/1000 Mbps), and most laptops implement a 54 Mbps Wireless Ethernet connection. Also, home wired or wireless LANs connecting all the home appliances, set-top boxes, desktops, and laptops to a shared Internet connection are very common. Spurgeon [2006] has provided a nice online summary of Ethernet technology, including some of its history.

System Area Networks

One of the first nonblocking multistage interconnection networks was proposed by Clos [1953] for use in telephone exchange offices. Building on this, many early inventions for system area networks came from their use in massively parallel processors (MPPs). One of the first MPPs was the Illiac IV, a SIMD array built in the early 1970s with 64 processing elements (“massive” at that time) interconnected using a topology based on a 2D torus that provided neighbor-to-neighbor communication. Another representative of early MPP was the Cosmic Cube, which used Ethernet interface chips to connect 64 processors in a 6-cube. Communication between nonneighboring nodes was made possible by store-and-forwarding of packets at intermediate nodes toward their final destination. A much larger and truly “massive” MPP built in the mid-1980s was the Connection Machine, a SIMD multiprocessor consisting of 64 K 1-bit processing elements, which also used a hypercube with store-and-forwarding. Since these early MPP machines, interconnection networks have improved considerably.

In the 1970s through the 1990s, considerable research went into trying to optimize the topology and, later, the routing algorithm, switching, arbitration, and flow control techniques. Initially, research focused on maximizing performance with

little attention paid to implementation constraints or crosscutting issues. Many exotic topologies were proposed having very interesting properties, but most of them complicated the routing. Rising from the fray was the hypercube, a very popular network in the 1980s that has all but disappeared from MPPs since the 1990s. What contributed to this shift was a performance model by Dally [1990] that showed that if the implementation is wire limited, lower-dimensional topologies achieve better performance than higher-dimensional ones because of their wider links for a given wire budget. Many designers followed that trend assuming their designs to be wire limited, even though most implementations were (and still are) pin limited. Several supercomputers since the 1990s have implemented low-dimensional topologies, including the Intel Paragon, Cray T3D, Cray T3E, HP AlphaServer, Intel ASCI Red, and IBM Blue Gene/L.

Meanwhile, other designers followed a very different approach, implementing bidirectional MINs in order to reduce the number of required switches below the number of network nodes. The most popular bidirectional MIN was the fat tree topology, originally proposed by Leiserson [1985] and first used in the Connection Machine CM-5 supercomputer and, later, the IBM ASCI White and ASC Purple supercomputers. This indirect topology was also used in several European parallel computers based on the Transputer. The Quadrics network has inherited characteristics from some of those Transputer-based networks. Myrinet has also evolved significantly from its first version, with Myrinet 2000 incorporating the fat tree as its principal topology. Indeed, most current implementations of SANs, including Myrinet, InfiniBand, and Quadrics as well as future implementations such as PCI-Express Advanced Switching, are based on fat trees.

Although the topology is the most visible aspect of a network, other features also have a significant impact on performance. A seminal work that raised awareness of deadlock properties in computer systems was published by Holt [1972]. Early techniques for avoiding deadlock in store-and-forward networks were proposed by Merlin and Schweitzer [1980] and by Gunther [1981]. Pipelined switching techniques were first introduced by Kermani and Kleinrock [1979] (virtual cut-through) and improved upon by Dally and Seitz [1986] (wormhole), which significantly reduced low-load latency and the topology's impact on message latency over previously proposed techniques. Wormhole switching was initially better than virtual cut-through largely because flow control could be implemented at a granularity smaller than a packet, allowing high-bandwidth links that were not as constrained by available switch memory bandwidth. Today, virtual cut-through is usually preferred over wormhole because it achieves higher throughput due to less HOL blocking effects and is enabled by current integration technology that allows the implementation of many packet buffers per link.

Tamir and Frazier [1992] laid the groundwork for virtual output queuing with the notion of dynamically allocated multiqueues. Around this same time, Dally [1992] contributed the concept of virtual channels, which was key to the development of more efficient deadlock-free routing algorithms and congestion-reducing flow control techniques for improved network throughput. Another highly relevant contribution to routing was a new theory proposed by Duato [1993] that allowed

the implementation of fully adaptive routing with just one “escape” virtual channel to avoid deadlock. Previous to this, the required number of virtual channels to avoid deadlock increased exponentially with the number of network dimensions. Pinkston and Warnakulasuriya [1997] went on to show that deadlock actually can occur very infrequently, giving credence to deadlock recovery routing approaches. Scott and Goodman [1994] were among the first to analyze the usefulness of pipelined channels for making link bandwidth independent of the time of flight. These and many other innovations have become quite popular, finding use in most high-performance interconnection networks, both past and present. The IBM Blue Gene/L, for example, implements virtual cut-through switching, four virtual channels per link, fully adaptive routing with one escape channel, and pipelined links.

MPPs represent a very small (and currently shrinking) fraction of the information technology market, giving way to bladed servers and clusters. In the United States, government programs such as the Advanced Simulation and Computing (ASC) program (formerly known as the Accelerated Strategic Computing Initiative, or ASCI) have promoted the design of those machines, resulting in a series of increasingly powerful one-of-a-kind MPPs costing \$50 million to \$100 million. These days, many are basically lower-cost clusters of symmetric multiprocessors (SMPs) (see Pfister [1998] and Sterling [2001] for two perspectives on clustering). In fact, in 2005, nearly 75% of the TOP500 supercomputers were clusters. Nevertheless, the design of each generation of MPPs and even clusters pushes interconnection network research forward to confront new problems arising due to sheer size and other scaling factors. For instance, source-based routing—the simplest form of routing—does not scale well to large systems. Likewise, fat trees require increasingly longer links as the network size increases, which led IBM Blue Gene/L designers to adopt a 3D torus network with distributed routing that can be implemented with bounded-length links.

Storage Area Networks

System area networks were originally designed for a single room or single floor (thus their distances are tens to hundreds of meters) and were for use in MPPs and clusters. In the intervening years, the acronym SAN has been co-opted to also mean storage area networks, whereby networking technology is used to connect storage devices to compute servers. Today, many refer to “storage” when they say SAN. The most widely used SAN example in 2006 was Fibre Channel (FC), which comes in many varieties, including various versions of Fibre Channel Arbitrated Loop (FC-AL) and Fibre Channel Switched (FC-SW). Not only are disk arrays attached to servers via FC links, but there are even some disks with FC links attached to switches so that storage area networks can enjoy the benefits of greater bandwidth and interconnectivity of switching.

In October 2000, the InfiniBand Trade Association announced the version 1.0 specification of InfiniBand [InfiniBand Trade Association 2001]. Led by Intel, HP, IBM, Sun, and other companies, it was targeted to the high-performance

computing market as a successor to the PCI bus by having point-to-point links and switches with its own set of protocols. Its characteristics are desirable potentially both for system area networks to connect clusters and for storage area networks to connect disk arrays to servers. Consequently, it has had strong competition from both fronts. On the storage area networking side, the chief competition for InfiniBand has been the rapidly improving Ethernet technology widely used in LANs. The Internet Engineering Task Force proposed a standard called iSCSI to send SCSI commands over IP networks [Satran et al. 2001]. Given the cost advantages of the higher-volume Ethernet switches and interface cards, Gigabit Ethernet dominates the low-end and medium range for this market. What's more, the slow introduction of InfiniBand and its small market share delayed the development of chip sets incorporating native support for InfiniBand. Therefore, network interface cards had to be plugged into the PCI or PCI-X bus, thus never delivering on the promise of replacing the PCI bus.

It was another I/O standard, PCI-Express, that finally replaced the PCI bus. Like InfiniBand, PCI-Express implements a switched network but with point-to-point serial links. To its credit, it maintains software compatibility with the PCI bus, drastically simplifying migration to the new I/O interface. Moreover, PCI-Express benefited significantly from mass market production and has found application in the desktop market for connecting one or more high-end graphics cards, making gamers very happy. Every PC motherboard now implements one or more 16x PCI-Express interfaces. PCI-Express absolutely dominates the I/O interface, but the current standard does not provide support for interprocessor communication.

Yet another standard, Advanced Switching Interconnect (ASI), may emerge as a complementary technology to PCI-Express. ASI is compatible with PCI-Express, thus linking directly to current motherboards, but it also implements support for interprocessor communication as well as I/O. Its defenders believe that it will eventually replace both SANs and LANs with a unified network in the data center market, but ironically this was also said of InfiniBand. The interested reader is referred to Pinkston et al. [2003] for a detailed discussion on this. There is also a new disk interface standard called Serial Advanced Technology Attachment (SATA) that is replacing parallel Integrated Device Electronics (IDE) with serial signaling technology to allow for increased bandwidth. Most disks in the market use this new interface, but keep in mind that Fibre Channel is still alive and well. Indeed, most of the promises made by InfiniBand in the SAN market were satisfied by Fibre Channel first, thus increasing their share of the market.

Some believe that Ethernet, PCI-Express, and SATA have the edge in the LAN, I/O interface, and disk interface areas, respectively. But the fate of the remaining storage area networking contenders depends on many factors. A wonderful characteristic of computer architecture is that such issues will not remain endless academic debates, unresolved as people rehash the same arguments repeatedly. Instead, the battle is fought in the marketplace, with well-funded and talented groups giving their best efforts at shaping the future. Moreover, constant changes to technology reward those who are either astute or lucky. The best combination of

technology and follow-through has often determined commercial success. Time will tell us who will win and who will lose, at least for the next round!

On-Chip Networks

Relative to the other network domains, on-chip networks are in their infancy. As recently as the late 1990s, the traditional way of interconnecting devices such as caches, register files, ALUs, and other functional units within a chip was to use dedicated links aimed at minimizing latency or shared buses aimed at simplicity. But with subsequent increases in the volume of interconnected devices on a single chip, the length and delay of wires to cross a chip, and chip power consumption, it has become important to share on-chip interconnect bandwidth in a more structured way, giving rise to the notion of a network on-chip. Among the first to recognize this were Agarwal [Waingold et al. 1997] and Dally [Dally 1999; Dally and Towles 2001]. They and others argued that on-chip networks that route packets allow efficient sharing of burgeoning wire resources between many communication flows and also facilitate modularity to mitigate chip-crossing wire delay problems identified by Ho, Mai, and Horowitz [2001]. Switched on-chip networks were also viewed as providing better fault isolation and tolerance. Challenges in designing these networks were later described by Taylor et al. [2005], who also proposed a 5-tuple model for characterizing the delay of OCNs. A design process for OCNs that provides a complete synthesis flow was proposed by Bertozzi et al. [2005]. Following these early works, much research and development has gone into on-chip network design, making this a very hot area of microarchitecture activity.

Multicore and tiled designs featuring on-chip networks have become very popular since the turn of the millennium. Pinkston and Shin [2005] provide a survey of on-chip networks used in early multicore/tiled systems. Most designs exploit the reduced wiring complexity of switched OCNs as the paths between cores/tiles can be precisely defined and optimized early in the design process, thus enabling improved power and performance characteristics. With typically tens of thousands of wires attached to the four edges of a core or tile as “pinouts,” wire resources can be traded off for improved network performance by having very wide channels over which data can be sent broadside (and possibly scaled up or down according to the power management technique), as opposed to serializing the data over fixed narrow channels.

Rings, meshes, and crossbars are straightforward to implement in planar chip technology and routing is easily defined on them, so these were popular topological choices in early switched OCNs. It will be interesting to see if this trend continues in the future when several tens to hundreds of heterogeneous cores and tiles will likely be interconnected within a single chip, possibly using 3D integration technology. Considering that processor microarchitecture has evolved significantly from its early beginnings in response to application demands and technological advancements, we would expect to see vast architectural improvements to on-chip networks as well.

References

- Agarwal, A., 1991. Limits on interconnection network performance. *IEEE Trans. on Parallel and Distributed Systems* 2 (4 (April)), 398–412.
- Alles, A., 1995. “ATM internetworking” (May). www.cisco.com/warp/public/614/12.html.
- Anderson, T.E., Culler, D.E., Patterson, D., 1995. A case for NOW (networks of workstations). *IEEE Micro* 15 (1 (February)), 54–64.
- Anjan, K.V., Pinkston, T.M., 1995. An efficient, fully-adaptive deadlock recovery scheme: Disha. In: Proc. 22nd Annual Int'l. Symposium on Computer Architecture, June 22–24, 1995. Santa Margherita Ligure, Italy.
- Arpacı, R.H., Culler, D.E., Krishnamurthy, A., Steinberg, S.G., Yellick, K., 1995. Empirical evaluation of the Cray-T3D: A compiler perspective. In: Proc. 22nd Annual Int'l. Symposium on Computer Architecture, June 22–24, 1995. Santa Margherita Ligure, Italy.
- Bell, G., Gray, J., 2001. Crays, Clusters and Centers. Microsoft Corporation, Redmond, Wash. MSR-TR-2001-76.
- Benes, V.E., 1962. Rearrangeable three stage connecting networks. *Bell Syst. Tech. J.* 41, 1481–1492.
- Bertozzi, D., Jalabert, A., Murali, S., Tamhankar, R., Stergiou, S., Benini, L., De Micheli, G., 2005. NoC synthesis flow for customized domain specific multiprocessor systems-on-chip. *IEEE Trans. on Parallel and Distributed Systems* 16 (2 (February)), 113–130.
- Bhuyan, L.N., Agrawal, D.P., 1984. Generalized hypercube and hyperbus structures for a computer network. *IEEE Trans. on Computers* 32 (4 (April)), 322–333.
- Brewer, E.A., Kuszmaul, B.C., 1994. How to get good performance from the CM-5 data network. In: Proc. Eighth Int'l Parallel Processing Symposium, April 26–29, 1994. Cancun, Mexico.
- Clos, C., 1953. A study of non-blocking switching networks. *Bell Systems Technical Journal* 32 (March), 406–424.
- Dally, W.J., 1990. Performance analysis of k-ary n-cube interconnection networks. *IEEE Trans. on Computers* 39 (6 (June)), 775–785.
- Dally, W.J., 1992. Virtual channel flow control. *IEEE Trans. on Parallel and Distributed Systems* 3 (2 (March)), 194–205.
- Dally, W.J., 1999. Interconnect limited VLSI architecture. In: Proc. of the Int'l. Interconnect Technology Conference, May 24–26, 1999. San Francisco, Calif.
- Dally, W.J., Seitz, C.I., 1986. The torus routing chip. *Distributed Computing* 1 (4), 187–196.
- Dally, W.J., Towles, B., 2001. Route packets, not wires: On-chip interconnection networks. In: Proc. of the 38th Design Automation Conference, June 18–22, 2001. Las Vegas, Nev.
- Dally, W.J., Towles, B., 2004. Principles and Practices of Interconnection Networks. Morgan Kaufmann Publishers, San Francisco.
- Davie, B.S., Peterson, L.L., Clark, D., 1999. Computer Networks: A Systems Approach, second ed. Morgan Kaufmann Publishers, San Francisco.
- Duato, J., 1993. A new theory of deadlock-free adaptive routing in wormhole networks. *IEEE Trans. on Parallel and Distributed Systems* 4 (12 (December)), 1320–1331.
- Duato, J., Pinkston, T.M., 2001. A general theory for deadlock-free adaptive routing using a mixed set of resources. *IEEE Trans. on Parallel and Distributed Systems* 12 (12 (December)), 1219–1235.
- Duato, J., Yalamanchili, S., Ni, L., 2003. Interconnection Networks: An Engineering Approach. Morgan Kaufmann Publishers, San Francisco. 2nd printing.
- Duato, J., Johnson, I., Fliech, J., Naven, F., Garcia, P., Nachiondo, T., 2005a. A new scalable and cost-effective congestion management strategy for lossless multistage interconnection networks. In: Proc. 11th Int'l. Symposium on High Performance Computer Architecture, February 12–16, 2005 San Francisco.
- Duato, J., Lysne, O., Pang, R., Pinkston, T.M., 2005b. Part I: A theory for deadlock-free dynamic reconfiguration of interconnection networks. *IEEE Trans. on Parallel and Distributed Systems* 16 (5 (May)), 412–427.
- Fliech, J., Bertozzi, D., 2010. Designing Network-on-Chip Architectures in the Nanoscale Era. CRC Press, Boca Raton, FL.
- Glass, C.J., Ni, L.M., 1992. The Turn Model for adaptive routing. In: Proc. 19th Int'l. Symposium on Computer Architecture, May, Gold Coast, Australia.
- Gunther, K.D., 1981. Prevention of deadlocks in packet-switched data transport systems. *IEEE Trans. on Communications*, 512–524. COM-29:4 (April).
- Ho, R., Mai, K.W., Horowitz, M.A., 2001. The future of wires. In: Proc. of the IEEE 89:4 (April), pp. 490–504.
- Holt, R.C., 1972. Some deadlock properties of computer systems. *ACM Computer Surveys* 4 (3 (September)), 179–196.

- Hoskote, Y., Vangal, S., Singh, A., Borkar, N., Borkar, S., 2007. A 5-ghz mesh interconnect for a teraflops processor. *IEEE Micro* 27 (5), 51–61.
- Howard, J., Dighe, S., Hoskote, Y., Vangal, S., Finan, S., Ruhl, G., Jenkins, D., Wilson, H., Borka, N., Schrom, G., Paillet, F., Jain, S., Jacob, T., Yada, S., Marella, S., Salihundam, P., Erraguntla, V., Konow, M., Riepen, M., Droege, G., Lindemann, J., Gries, M., Apel, T., Henriss, K., Lund-Larsen, T., Steibl, S., Borkar, S., De, V., Van Der Wijngaart, R., Mattson, T., 2010. A 48-core IA-32 message-passing processor with DVFS in 45 nm CMOS. In: *IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pp. 58–59.
- InfiniBand Trade Association, 2001. InfiniBand Architecture Specifications Release 1.0.a. www.infinibandta.org.
- Jantsch, A., Tenhunen, H. (Eds.), 2003. *Networks on Chips*. Kluwer Academic Publishers, The Netherlands.
- Kahn, R.E., 1972. Resource-sharing computer communication networks. In: *Proc. IEEE* 60:11 (November), pp. 1397–1407.
- Kermani, P., Kleinrock, L., 1979. Virtual cut-through: A new computer communication switching technique. *Computer Networks* 3 (January), 267–286.
- Kurose, J.F., Ross, K.W., 2001. *Computer Networking: A Top-Down Approach Featuring the Internet*. Addison-Wesley, Boston.
- Leiserson, C.E., 1985. Fat trees: Universal networks for hardware-efficient supercomputing. *IEEE Trans. on Computers*, 892–901. C-34:10 (October).
- Merlin, P.M., Schweitzer, P.J., 1980. Deadlock avoidance in store-and-forward networks. I. Store-and-forward deadlock. *IEEE Trans. on Communications*, 345–354. COM-28:3 (March).
- Metcalfe, R.M., 1993. Computer/network interface design: Lessons from Arpanet and Ethernet. *IEEE J. on Selected Areas in Communications* 11 (2 (February)), 173–180.
- Metcalfe, R.M., Boggs, D.R., 1976. Ethernet: Distributed packet switching for local computer networks. *Comm. ACM* 19 (7 (July)), 395–404.
- Partridge, C., 1994. *Gigabit Networking*. Addison-Wesley, Reading, Mass.
- Peh, L.S., Dally, W.J., 2001. A delay model and speculative architecture for pipelined routers. In: *Proc. 7th Int'l. Symposium on High Performance Computer Architecture*, January 20–24, 2001. Monterey, Mexico.
- Pfister, G.F., 1998. *In Search of Clusters*, second ed. Prentice Hall, Upper Saddle River, N.J.
- Pinkston, T.M., 2004. Deadlock characterization and resolution in interconnection networks. In: Zhu, M.C., Fanti, M.P. (Eds.), *Deadlock Resolution in Computer-Integrated Systems*. CRC Press, Boca Raton, Fl, pp. 445–492.
- Pinkston, T.M., Shin, J., 2005. Trends toward on-chip networked microsystems. *Int'l. J. of High Performance Computing and Networking* 3 (1), 3–18.
- Pinkston, T.M., Warnakulasuriya, S., 1997. On deadlocks in interconnection networks. In: *Proc. 24th Int'l. Symposium on Computer Architecture*, June 2–4, 1997. Denver, Colo.
- Pinkston, T.M., Benner, A., Krause, M., Robinson, I., Sterling, T., 2003. InfiniBand: The ‘de facto’ future standard for systems and local area networks or just a scalable replacement for PCI buses? “Special Issue on Communication Architecture for Clusters” 6:2 (April). *Cluster Computing*, 95–104.
- Puente, V., Beivide, R., Gregorio, J.A., Prellezo, J.M., Duato, J., Izu, C., 1999. Adaptive bubble router: A design to improve performance in torus networks. In: *Proc. 28th Int'l. Conference on Parallel Processing*, September 21–24, 1999. Aizu-Wakamatsu, Japan.
- Rodrigo, S., Flieh, J., Duato, J., Hummel, M., 2008. Efficient unicast and multicast support for CMPs. In: *Proc. 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-41)*, November 8–12, 2008. Lake Como, Italy, pp. 364–375.
- Saltzer, J.H., Reed, D.P., Clark, D.D., 1984. End-to-end arguments in system design. *ACM Trans. on Computer Systems* 2 (4 (November)), 277–288.
- Satran, J., Smith, D., Meth, K., Sapuntzakis, C., Wakeley, M., Von Stamwitz, P., Haagens, R., Zeidner, E., Dalle Ore, L., Klein, Y., 2001. “iSCSI”, IPS working group of IETF, Internet draft. www.ietf.org/internet-drafts/draft-ietf-ips-iscsi-07.txt.
- Scott, S.L., Goodman, J., 1994. The impact of pipelined channels on k-ary n-cube networks. *IEEE Trans. on Parallel and Distributed Systems* 5 (1 (January)), 1–16.
- Senior, J.M., 1993. *Optical Fiber Communications: Principles and Practice*, second ed. Prentice Hall, Hertfordshire, U.K..
- Spurgeon, C., 2006. Charles Spurgeon’s Ethernet Web Site. www.etherman-age.com/ethernet/ethernet.html.

- Sterling, T., 2001. Beowulf PC Cluster Computing with Windows and Beowulf PC Cluster Computing with Linux. MIT Press, Cambridge, Mass.
- Stevens, W.R., 1994–1996. *TCP/IP Illustrated* (three volumes). Addison-Wesley, Reading, Mass.
- Tamir, Y., Frazier, G., 1992. Dynamically-allocated multi-queue buffers for VLSI communication switches. *IEEE Trans. on Computers* 41 (6 (June)), 725–734.
- Tanenbaum, A.S., 1988. Computer Networks, second ed. Prentice Hall, Englewood Cliffs, N.J.
- Taylor, M.B., Lee, W., Amarasinghe, S.P., Agarwal, A., 2005. Scalar operand networks. *IEEE Trans. on Parallel and Distributed Systems* 16 (2 (February)), 145–162.
- Thacker, C.P., McCreight, E.M., Lampson, B.W., Sproull, R.F., Boggs, D.R., 1982. Alto: A personal computer. In: Siewiorek, D.P., Bell, C.G., Newell, A. (Eds.), *Computer Structures: Principles and Examples*. McGraw-Hill, New York, pp. 549–572.
- TILE-GX, http://www.tilera.com/sites/default/files/productbriefs/PB025_TILE-Gx_Processor_A_v3.pdf.
- Vaidya, A.S., Sivasubramaniam, A., Das, C.R., 1997. Performance benefits of virtual channels and adaptive routing: An application-driven study. In: Proc. 11th ACM Int'l Conference on Supercomputing, July 7–11, 1997. Vienna, Austria.
- Van Leeuwen, J., Tan, R.B., 1987. Interval Routing. *The Computer Journal* 30 (4), 298–307.
- von Eicken, T., Culler, D.E., Goldstein, S.C., Schauer, K.E., 1992. Active messages: A mechanism for integrated communication and computation. In: Proc. 19th Annual Int'l. Symposium on Computer Architecture, May 19–21, 1992. Gold Coast, Australia.
- Waingold, E., Taylor, M., Srikrishna, D., Sarkar, V., Lee, W., Lee, V., Kim, J., Frank, M., Finch, P., Barua, R., Babb, J., Amarasinghe, S., Agarwal, A., 1997. Baring it all to software: Raw Machines. *IEEE Computer* 30 (September), 86–93.
- Yang, Y., Mason, G., 1991. Nonblocking broadcast switching networks. *IEEE Trans. on Computers* 40 (9 (September)), 1005–1015.

Exercises

Solutions to “starred” exercises are available for instructors who register at *text-books.elsevier.com*.

- ★ F.1 [15]<F.2, F.3>Is electronic communication always faster than nonelectronic means for longer distances? Calculate the time to send 1000 GB using 25 8-mm tapes and an overnight delivery service versus sending 1000 GB by FTP over the Internet. Make the following four assumptions:
- The tapes are picked up at 4 P.M. Pacific time and delivered 4200 km away at 10 A.M. Eastern time (7 A.M. Pacific time).
 - On one route the slowest link is a T3 line, which transfers at 45 Mbits/sec.
 - On another route the slowest link is a 100-Mbit/sec Ethernet.
 - You can use 50% of the slowest link between the two sites.
- Will all the bytes sent by either Internet route arrive before the overnight delivery person arrives?
- ★ F.2 [10]<F.2, F.3>For the same assumptions as Exercise F.1, what is the bandwidth of overnight delivery for a 1000-GB package?
- ★ F.3 [10]<F.2, F.3>For the same assumptions as Exercise F.1, what is the minimum bandwidth of the slowest link to beat overnight delivery? What standard network options match that speed?

- ★ F.4 [15]<F.2, F.3> The original Ethernet standard was for 10 Mbits/sec and a maximum distance of 2.5 km. How many bytes could be in flight in the original Ethernet? Assume you can use 90% of the peak bandwidth.
- ★ F.5 [15]<F.2, F.3> Flow control is a problem for WANs due to the long time of flight, as the example on page F-14 illustrates. Ethernet did not include flow control when it was first standardized at 10 Mbits/sec. Calculate the number of bytes in flight for a 10-Gbit/sec Ethernet over a 100 meter link, assuming you can use 90% of peak bandwidth. What does your answer mean for network designers?
- ★ F.6 [15]<F.2, F.3> Assume the total overhead to send a zero-length data packet on an Ethernet is 100 μ s and that an unloaded network can transmit at 90% of the peak 1000-Mbit/sec rating. For the purposes of this question, assume that the size of the Ethernet header and trailer is 56 bytes. Assume a continuous stream of packets of the same size. Plot the delivered bandwidth of user data in Mbits/sec as the payload data size varies from 32 bytes to the maximum size of 1500 bytes in 32-byte increments.
- ★ F.7 [10]<F.2, F.3> Exercise F.6 suggests that the delivered Ethernet bandwidth to a single user may be disappointing. Making the same assumptions as in that exercise, by how much would the maximum payload size have to be increased to deliver half of the peak bandwidth?
- ★ F.8 [10]<F.2, F.3> One reason that ATM has a fixed transfer size is that when a short message is behind a long message, a node may need to wait for an entire transfer to complete. For applications that are time sensitive, such as when transmitting voice or video, the large transfer size may result in transmission delays that are too long for the application. On an unloaded interconnection, what is the worstcase delay in microseconds if a node must wait for one full-size Ethernet packet versus an ATM transfer? See Figure F.30 (page F-78) to find the packet sizes. For this question assume that you can transmit at 100% of the 622-Mbits/sec ATM network and 100% of the 1000-Mbit/sec Ethernet.
- ★ F.9 [10]<F.2, F.3> Exercise F.7 suggests the need for expanding the maximum pay-load to increase the delivered bandwidth, but Exercise F.8 suggests the impact on worst-case latency of making it longer. What would be the impact on latency of increasing the maximum payload size by the answer to Exercise F.7?
- ★ F.10 [12/12/20]<F.4> The Omega network shown in Figure F.11 on page F-31 consists of three columns of four switches, each with two inputs and two outputs. Each switch can be set to *straight*, which connects the upper switch input to the upper switch output and the lower input to the lower output, and to *exchange*, which connects the upper input to the lower output and *vice versa* for the lower input. For each column of switches, label the inputs and outputs 0, 1, ..., 7 from top to bottom, to correspond with the numbering of the processors.

- a. [12]<F.4> When a switch is set to exchange and a message passes through, what is the relationship between the label values for the switch input and output used by the message? (*Hint:* Think in terms of operations on the digits of the binary representation of the label number.)
 - b. [12]<F.4> Between any two switches in adjacent columns that are connected by a link, what is the relationship between the label of the output connected to the input?
 - c. [20]<F.4> Based on your results in parts (a) and (b), design and describe a simple routing scheme for distributed control of the Omega network. A message will carry a *routing tag* computed by the sending processor. Describe how the processor computes the tag and how each switch can set itself by examining a bit of the routing tag.
- ★ F.11 [12/12/12/12/12/12]<F.4> Prove whether or not it is possible to realize the following permutations (i.e., communication patterns) on the eight-node Omega network shown in Figure F.11 on page F-31:
- a. [12]<F.4> Bit-reversal permutation—the node with binary coordinates $a_{n-1}, a_{n-2}, \dots, a_1, a_0$ communicates with the node $a_0, a_1, \dots, a_{n-2}, a_{n-1}$.
 - b. [12]<F.4> Perfect shuffle permutation—the node with binary coordinates $a_{n-1}, a_{n-2}, \dots, a_1, a_0$ communicates with the node $a_{n-2}, a_{n-3}, \dots, a_0, a_{n-1}$ (i.e., rotate left 1 bit).
 - c. [12]<F.4> Bit-complement permutation—the node with binary coordinates $a_{n-1}, a_{n-2}, \dots, a_1, a_0$ communicates with the node $\bar{a}_{n-1}, \bar{a}_{n-2}, \dots, \bar{a}_1, \bar{a}_0$ (i.e., complement each bit).
 - d. [12]<F.4> Butterfly permutation—the node with binary coordinates $a_{n-1}, a_{n-2}, \dots, a_1, a_0$ communicates with the node $a_0, a_{n-2}, \dots, a_1, a_{n-1}$ (i.e., swap the most and least significant bits).
 - e. [12]<F.4> Matrix transpose permutation—the node with binary coordinates $a_{n-1}, a_{n-2}, \dots, a_1, a_0$ communicates with the node $a_{n/2-1}, \dots, a_0, a_{n-1}, \dots, a_{n/2}$ (i.e., transpose the bits in positions approximately halfway around).
 - f. [12]<F.4> Barrel-shift permutation—node i communicates with node $i+1$ modulo $N-1$, where N is the total number of nodes and $0 \leq i$.
- ★ F.12 [12]<F.4> Design a network topology using 18-port crossbar switches that has the minimum number of switches to connect 64 nodes. Each switch port supports communication to and from one device.
- ★ F.13 [15]<F.4> Design a network topology that has the minimum latency through the switches for 64 nodes using 18-port crossbar switches. Assume unit delay in the switches and zero delay for wires.
- ★ F.14 [15]<F.4> Design a switch topology that balances the bandwidth required for all links for 64 nodes using 18-port crossbar switches. Assume a uniform traffic pattern.

- ★ F.15 [15]<F.4> Compare the interconnection latency of a crossbar, Omega network, and fat tree with eight nodes. Use Figure F.11 on page F-31, Figure F.12 on page F-33, and Figure F.14 on page F-37. Assume that the fat tree is built entirely from two-input, two-output switches so that its hardware resources are more comparable to that of the Omega network. Assume that each switch costs a unit time delay. Assume that the fat tree randomly picks a path, so give the best case and worst case for each example. How long will it take to send a message from node 0 to node 6? How long will it take node 1 and node 7 to communicate?
- ★ F.16 [15]<F.4> Draw the topology of a 6-cube after the same manner of the 4-cube in Figure F.14 on page F-37. What is the maximum and average number of hops needed by packets assuming a uniform distribution of packet destinations?
- ★ F.17 [15]<F.4> Complete a table similar to Figure F.15 on page F-40 that captures the performance and cost of various network topologies, but do it for the general case of N nodes using $k \times k$ switches instead of the specific case of 64 nodes.
- ★ F.18 [20]<F.4> Repeat the example given on page F-41, but use the bit-complement communication pattern given in Exercise F.11 instead of NEWS communication.
- ★ F.19 [15]<F.5> Give the four specific conditions necessary for deadlock to exist in an interconnection network. Which of these are removed by dimension-order routing? Which of these are removed in adaptive routing with the use of “escape” routing paths? Which of these are removed in adaptive routing with the technique of deadlock recovery (regressive or progressive)? Explain your answer.
- ★ F.20 [12/12/12/12]<F.5> Prove whether or not the following routing algorithms based on prohibiting dimensional turns are suitable to be used as escape paths for 2D meshes by analyzing whether they are both connected and deadlock-free. Explain your answer. (*Hint:* You may wish to refer to the Turn Model algorithm and/or to prove your answer by drawing a directed graph for a 4×4 mesh that depicts dependencies between channels and verifying the channel dependency graph is free of cycles.) The routing algorithms are expressed with the following abbreviations: W = west, E = east, N = north, and S = south.
- [12]<F.5> Allowed turns are from W to N, E to N, S to W, and S to E.
 - [12]<F.5> Allowed turns are from W to S, E to S, N to E, and S to E.
 - [12]<F.5> Allowed turns are from W to S, E to S, N to W, S to E, W to N, and S to W.
 - [12]<F.5> Allowed turns are from S to E, E to S, S to W, N to W, N to E, and E to N.
- ★ F.21 [15]<F.5> Compute and compare the upper bound for the efficiency factor, ρ , for dimension-order routing and up*/down* routing assuming uniformly distributed traffic on a 64-node 2D mesh network. For up*/down* routing, assume optimal placement of the root node (i.e., a node near the middle of the mesh). (*Hint:* You will have to find the loading of links across the network bisection that carries the global load as determined by the routing algorithm.)

- ★ F.22 [15]<F.5>For the same assumptions as Exercise F.21, find the efficiency factor for up*/down* routing on a 64-node fat tree network using 4×4 switches. Compare this result with the ρ found for up*/down* routing on a 2D mesh. Explain.
- ★ F.23 [15]<F.5>Calculate the probability of matching two-phased arbitration requests from all k input ports of a switch simultaneously to the k output ports assuming a uniform distribution of requests and grants to/from output ports. How does this compare to the matching probability for three-phased arbitration in which each of the k input ports can make two simultaneous requests (again, assuming a uniform random distribution of requests and grants)?
- ★ F.24 [15]<F.5>The equation on page F-52 shows the value of cut-through switching. Ethernet switches used to build clusters often do not support cut-through switching. Compare the time to transfer 1500 bytes over a 1000-Mbit/sec Ethernet with and without cut-through switching for a 64-node cluster. Assume that each Ethernet switch takes 1.0 μ s and that a message goes through seven intermediate switches.
- ★ F.25 [15]<F.5>Making the same assumptions as in Exercise F.24, what is the difference between cut-through and store-and-forward switching for 32 bytes?
- ★ F.26 [15]<F.5>One way to reduce latency is to use larger switches. Unlike Exercise F.24, let's assume we need only three intermediate switches to connect any two nodes in the cluster. Make the same assumptions as in Exercise F.24 for the remaining parameters. What is the difference between cut-through and store-and-forward for 1500 bytes? For 32 bytes?
- ★ F.27 [20]<F.5>Using FlexSim 1.2 (<http://ceng.usc.edu/smarr/FlexSim/flexsim.html>) or some other cycle-accurate network simulator, simulate a 256-node 2D torus network assuming wormhole routing, 32-flit packets, uniform (random) communication pattern, and four virtual channels. Compare the performance of deterministic routing using DOR, adaptive routing using escape paths (i.e., Duato's Protocol), and true fully adaptive routing using progressive deadlock recovery (i.e., Disha routing). Do so by plotting latency versus applied load and through-put versus applied load for each, as is done in Figure F.19 for the example on page F-53. Also run simulations and plot results for two and eight virtual channels for each. Compare and explain your results by addressing how/why the number and use of virtual channels by the various routing algorithms affect network performance. (*Hint:* Be sure to let the simulation reach steady state by allowing a warm-up period of a several thousand network cycles before gathering results.)
- ★ F.28 [20]<F.5>Repeat Exercise F.27 using bit-reversal communication instead of the uniform random communication pattern. Compare and explain your results by addressing how/why the communication pattern affects network performance.
- ★ F.29 [40]<F.5>Repeat Exercises F.27 and F.28 using 16-flit packets and 128-flit packets. Compare and explain your results by addressing how/why the packet size along with the other design parameters affect network performance.
- F.30 [20]<F.2, F.4, F.5, F.8>Figures F.7, F.16, and F.20 show interconnection network characteristics of several of the top 500 supercomputers by machine type

as of the publication of the fourth edition. Update that figure to the most recent top 500. How have the systems and their networks changed since the data in the original figure? Do similar comparisons for OCNs used in microprocessors and SANs targeted for clusters using Figures F.29 and F.31.

- ★ F.31 [12/12/12/15/18] <F.8> Use the M/M/1 queuing model to answer this exercise. Measurements of a network bridge show that packets arrive at 200 packets per second and that the gateway forwards them in about 2 ms.
 - a. [12] <F.8> What is the utilization of the gateway?
 - b. [12] <F.8> What is the mean number of packets in the gateway?
 - c. [12] <F.8> What is the mean time spent in the gateway?
 - d. [15] <F.8> Plot response time versus utilization as you vary the arrival rate.
 - e. [15] <F.8> For an M/M/1 queue, the probability of finding n or more tasks in the system is Utilization n . What is the chance of an overflow of the FIFO if it can hold 10 messages?
 - f. [18] <F.8> How big must the gateway be to have packet loss due to FIFO overflow less than one packet per million?
- ★ F.32 [20] <F.8> The imbalance between the time of sending and receiving can cause problems in network performance. Sending too fast can cause the network to back up and increase the latency of messages, since the receivers will not be able to pull out the message fast enough. A technique called *bandwidth matching* proposes a simple solution: Slow down the sender so that it matches the performance of the receiver [Brewer and Kuszmaul 1994]. If two machines exchange an equal number of messages using a protocol like UDP, one will get ahead of the other, causing it to send all its messages first. After the receiver puts all these messages away, it will then send its messages. Estimate the performance for this case versus a bandwidth-matched case. Assume that the send overhead is 200 μ s, the receive overhead is 300 μ s, time of flight is 5 μ s, latency is 10 μ s, and that the two machines want to exchange 100 messages.
- F.33 [40] <F.8> Compare the performance of UDP with and without bandwidth matching by slowing down the UDP send code to match the receive code as advised by bandwidth matching [Brewer and Kuszmaul 1994]. Devise an experiment to see how much performance changes as a result. How should you change the send rate when two nodes send to the same destination? What if one sender sends to two destinations?
- ★ F.34 [40] <F.6, F.8> If you have access to an SMP and a cluster, write a program to measure latency of communication and bandwidth of communication between processors, as was plotted in Figure F.32 on page F-80.
- F.35 [20/20/20] <F.9> If you have access to a UNIX system, use ping to explore the Internet. First read the manual page. Then use ping without option flags to be sure you can reach the following sites. It should say that X is alive. Depending on your system, you may be able to see the path by setting the flags to verbose mode

(*-v*) and trace route mode (*-R*) to see the path between your machine and the example machine. Alternatively, you may need to use the program `trace route` to see the path. If so, try its manual page. You may want to use the UNIX command `script` to make a record of your session.

- a. [20]<F.9>Trace the route to another machine on the same local area network. What is the latency?

- b. [20]<F.9>Trace the route to another machine on your campus that is *not* on the same local area network.What is the latency?

- c. [20]<F.9>Trace the route to another machine *off campus*. For example, if you have a friend you send email to, try tracing that route. See if you can discover what types of networks are used along that route.What is the latency?

F.36 [15]<F.9>Use FTP to transfer a file from a remote site and then between local sites on the same LAN. What is the difference in bandwidth for each transfer? Try the transfer at different times of day or days of the week. Is the WAN or LAN the bottleneck?

★ F.37 [10/10]<F.9, F.11>Figure F.41 on page F-93 compares latencies for a high-bandwidth network with high overhead and a low-bandwidth network with low overhead for different TCP/IP message sizes.

- a. [10]<F.9, F.11>For what message sizes is the delivered bandwidth higher for the high-bandwidth network?

- b. [10]<F.9, F.11>For your answer to part (a), what is the delivered bandwidth for each network?

★ F.38 [15]<F.9, F.11>Using the statistics in Figure F.41 on page F-93, estimate the per-message overhead for each network.

★ F.39 [15]<F.9, F.11>Exercise F.37 calculates which message sizes are faster for two networks with different overhead and peak bandwidth. Using the statistics in Figure F.41 on page F-93, what is the percentage of messages that are transmitted more quickly on the network with low overhead and bandwidth? What is the percentage of data transmitted more quickly on the network with high overhead and bandwidth?

★ F.40 [15]<F.9, F.11>One interesting measure of the latency and bandwidth of an inter-connection is to calculate the size of a message needed to achieve one-half of the peak bandwidth. This halfway point is sometimes referred to as $n_{1/2}$, taken from the terminology of vector processing. Using Figure F.41 on page F-93, estimate $n_{1/2}$ for TCP/IP message using 155-Mbit/sec ATM and 10-Mbit/sec Ethernet.

F.41 [Discussion]<F.10>The Google cluster used to be constructed from 1 rack unit (RU) PCs, each with one processor and two disks. Today there are considerably denser options. How much less floor space would it take if we were to replace the 1 RU PCs with modern alternatives? Go to the Compaq or Dell Web sites to find the densest alternative. What would be the estimated impact on cost of the equipment? What would be the estimated impact on rental cost of floor space?

What would be the impact on interconnection network design for achieving power/performance efficiency?

- F.42 [Discussion] <F.13> At the time of the writing of the fourth edition, it was unclear what would happen with Ethernet versus InfiniBand versus Advanced Switching in the machine room. What are the technical advantages of each? What are the economic advantages of each? Why would people maintaining the system prefer one to the other? How popular is each network today? How do they compare to proprietary commercial networks such as Myrinet and Quadrics?

G.1	Introduction	G-2
G.2	Vector Performance in More Depth	G-2
G.3	Vector Memory Systems in More Depth	G-9
G.4	Enhancing Vector Performance	G-11
G.5	Effectiveness of Compiler Vectorization	G-14
G.6	Putting It All Together: Performance of Vector Processors	G-15
G.7	A Modern Vector Supercomputer: The Cray X1	G-21
G.8	Concluding Remarks	G-25
G.9	Historical Perspective and References	G-26
	Exercises	G-29

G

Vector Processors in More Depth

**Revised by Krste Asanovic
Massachusetts Institute of Technology**

I'm certainly not inventing vector processors. There are three kinds that I know of existing today. They are represented by the Illiac-IV, the (CDC) Star processor, and the TI (ASC) processor. Those three were all pioneering processors....One of the problems of being a pioneer is you always make mistakes and I never, never want to be a pioneer. It's always best to come second when you can look at the mistakes the pioneers made.

Seymour Cray
Public lecture at Lawrence Livermore Laboratories on the introduction of the Cray-1 (1976)

G.1**Introduction**

Chapter 4 introduces vector architectures and places Multimedia SIMD extensions and GPUs in proper context to vector architectures.

In this appendix, we go into more detail on vector architectures, including more accurate performance models and descriptions of previous vector architectures. Figure G.1 shows the characteristics of some typical vector processors, including the size and count of the registers, the number and types of functional units, the number of load-store units, and the number of lanes.

G.2**Vector Performance in More Depth**

The chime approximation is reasonably accurate for long vectors. Another source of overhead is far more significant than the issue limitation.

The most important source of overhead ignored by the chime model is vector *start-up time*. The start-up time comes from the pipelining latency of the vector operation and is principally determined by how deep the pipeline is for the functional unit used. The start-up time increases the effective time to execute a convoy to more than one chime. Because of our assumption that convoys do not overlap in time, the start-up time delays the execution of subsequent convoys. Of course, the instructions in successive convoys either have structural conflicts for some functional unit or are data dependent, so the assumption of no overlap is reasonable. The actual time to complete a convoy is determined by the sum of the vector length and the start-up time. If vector lengths were infinite, this start-up overhead would be amortized, but finite vector lengths expose it, as the following example shows.

Example Assume that the start-up overhead for functional units is shown in Figure G.2.

Show the time that each convoy can begin and the total number of cycles needed. How does the time compare to the chime approximation for a vector of length 64?

Answer

Figure G.3 provides the answer in convoys, assuming that the vector length is n . One tricky question is when we assume the vector sequence is done; this determines whether the start-up time of the SV is visible or not. We assume that the instructions following cannot fit in the same convoy, and we have already assumed that convoys do not overlap. Thus, the total time is given by the time until the last vector instruction in the last convoy completes. This is an approximation, and the start-up time of the last vector instruction may be seen in some sequences and not in others. For simplicity, we always include it.

The time per result for a vector of length 64 is $4 + (42/64) = 4.65$ clock cycles, while the chime approximation would be 4. The execution time with startup overhead is 1.16 times higher.

Processor (year)	Vector clock rate (MHz)	Vector registers	Elements per register (64-bit elements)	Vector arithmetic units	Vector load-store units	Lanes
Cray-1 (1976)	80	8	64	6: FP add, FP multiply, FP reciprocal, integer add, logical, shift	1	1
Cray X-MP (1983)	118	8	64	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	2 loads 1 store	1
Cray Y-MP (1988)	166					
Cray-2 (1985)	244	8	64	5: FP add, FP multiply, FP reciprocal/sqrt, integer addshift/population count, logical	1	1
Fujitsu VP100/VP200 (1982)	133	8–256	32–1024	3: FP or integer addlogical, multiply, divide	2 2 (VP200)	1 (VP100) 2 (VP200)
Hitachi S810/S820 (1983)	71	32	256	4: FP multiply-add, FP multiply/divide-add unit, 2 integer addlogical	3 loads 1 store	1 (S810) 2 (S820)
Convex C-1 (1985)	10	8	128	2: FP or integer multiply/divide, addlogical	1 2 (32 bit)	1 (64 bit) 2 (32 bit)
NEC SX/2 (1985)	167	8+32	256	4: FP multiply/divide, FP add, integer addlogical, shift	1	4
Cray C90 (1991)	240	8	128	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	2 loads 1 store	2
Cray T90 (1995)	460					
NEC SX/5 (1998)	312	8+64	512	4: FP or integer addshift, multiply, divide, logical	1	16
Fujitsu VPP5000 (1999)	300	8–256	128–4096	3: FP or integer multiply, addlogical, divide	1 load 1 store	16
Cray SV1 (1998)	300	8	64 (MSP)	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	1 load-store 1 load	2 8 (MSP)
SV1ex (2001)	500					
VMIPS (2001)	500	8	64	5: FP multiply, FP divide, FP add, integer addshift, logical	1 load-store	1
NEC SX/6 (2001)	500	8+64	256	4: FP or integer addshift, multiply, divide, logical	1	8
NEC SX/8 (2004)	2000	8+64	256	4: FP or integer addshift, multiply, divide, logical	1	4
Cray X1 (2002)	800	32	64	3: FP or integer, addlogical, multiplyshift, divide/square root/logical	1 load 1 store	2 8 (MSP)
Cray XIE (2005)	1130		256 (MSP)			

Figure G.1 Characteristics of several vector-register architectures. If the machine is a multiprocessor, the entries correspond to the characteristics of one processor. Several of the machines have different clock rates in the vector and scalar units; the clock rates shown are for the vector units. The Fujitsu machines' vector registers are configurable: The size and count of the 8K 64-bit entries may be varied inversely to one another (e.g., on the VP200, from eight registers each 1K elements long to 256 registers each 32 elements long). The NEC machines have eight foreground vector registers connected to the arithmetic units plus 32 to 64 background vector registers connected between the memory system and the foreground vector registers. Add pipelines perform add and subtract. The multiply/divide-add unit on the Hitachi S810/S820 performs an FP multiply or divide followed by an add or subtract (while the multiply-add unit performs a multiply followed by an add or subtract). Note that most processors use the vector FP multiply and divide units for vector integer multiply and divide, and several of the processors use the same units for FP scalar and FP vector operations. Each vector load-store unit represents the ability to do an independent, overlapped transfer to or from the vector registers. The number of lanes is the number of parallel pipelines in each of the functional units as described in Section G.4. For example, the NEC SX/5 can complete 16 multiplies per cycle in the multiply functional unit. Several machines can split a 64-bit lane into two 32-bit lanes to increase performance for applications that require only reduced precision. The Cray SV1 and Cray X1 can group four CPUs with two lanes each to act in unison as a single larger CPU with eight lanes, which Cray calls a Multi-Streaming Processor (MSP).

Unit	Start-up overhead (cycles)
Load and store unit	12
Multiply unit	7
Add unit	6

Figure G.2 Start-up overhead.

Convoy	Starting time	First-result time	Last-result time
1. LV	0	12	$11+n$
2. MULVS.D LV	$12+n$	$12+n+12$	$23+2n$
3. ADDV.D	$24+2n$	$24+2n+6$	$29+3n$
4. SV	$30+3n$	$30+3n+12$	$41+4n$

Figure G.3 Starting times and first- and last-result times for convoys 1 through 4. The vector length is n .

For simplicity, we will use the chime approximation for running time, incorporating start-up time effects only when we want performance that is more detailed or to illustrate the benefits of some enhancement. For long vectors, a typical situation, the overhead effect is not that large. Later in the appendix, we will explore ways to reduce start-up overhead.

Start-up time for an instruction comes from the pipeline depth for the functional unit implementing that instruction. If the initiation rate is to be kept at 1 clock cycle per result, then

$$\text{Pipeline depth} = \left\lceil \frac{\text{Total functional unit time}}{\text{Clock cycle time}} \right\rceil$$

For example, if an operation takes 10 clock cycles, it must be pipelined 10 deep to achieve an initiation rate of one per clock cycle. Pipeline depth, then, is determined by the complexity of the operation and the clock cycle time of the processor. The pipeline depths of functional units vary widely—2 to 20 stages are common—although the most heavily used units have pipeline depths of 4 to 8 clock cycles.

For VMIPS, we will use the same pipeline depths as the Cray-1, although latencies in more modern processors have tended to increase, especially for loads. All functional units are fully pipelined. From Chapter 4, pipeline depths are 6 clock cycles for floating-point add and 7 clock cycles for floating-point multiply. On VMIPS, as on most vector processors, independent vector operations using different functional units can issue in the same convoy.

In addition to the start-up overhead, we need to account for the overhead of executing the strip-mined loop. This strip-mining overhead, which arises from

Operation	Start-up penalty
Vector add	6
Vector multiply	7
Vector divide	20
Vector load	12

Figure G.4 Start-up penalties on VMIPS. These are the start-up penalties in clock cycles for VMIPS vector operations.

the need to reinitiate the vector sequence and set the Vector Length Register (VLR) effectively adds to the vector start-up time, assuming that a convoy does not overlap with other instructions. If that overhead for a convoy is 10 cycles, then the effective overhead per 64 elements increases by 10 cycles, or 0.15 cycles per element.

Two key factors contribute to the running time of a strip-mined loop consisting of a sequence of convoys:

1. The number of convoys in the loop, which determines the number of chimes. We use the notation T_{chime} for the execution time in chimes.
2. The overhead for each strip-mined sequence of convoys. This overhead consists of the cost of executing the scalar code for strip-mining each block, T_{loop} , plus the vector start-up cost for each convoy, T_{start} .

There may also be a fixed overhead associated with setting up the vector sequence the first time. In recent vector processors, this overhead has become quite small, so we ignore it.

The components can be used to state the total running time for a vector sequence operating on a vector of length n , which we will call T_n :

$$T_n = \left[\frac{n}{\text{MVL}} \right] \times (T_{\text{loop}} + T_{\text{start}}) + n \times T_{\text{chime}}$$

The values of T_{start} , T_{loop} , and T_{chime} are compiler and processor dependent. The register allocation and scheduling of the instructions affect both what goes in a convoy and the start-up overhead of each convoy.

For simplicity, we will use a constant value for T_{loop} on VMIPS. Based on a variety of measurements of Cray-1 vector execution, the value chosen is 15 for T_{loop} . At first glance, you might think that this value is too small. The overhead in each loop requires setting up the vector starting addresses and the strides, incrementing counters, and executing a loop branch. In practice, these scalar instructions can be totally or partially overlapped with the vector instructions, minimizing the time spent on these overhead functions. The value of T_{loop} of course depends on the loop structure, but the dependence is slight compared with the connection between the vector code and the values of T_{chime} and T_{start} .

Example What is the execution time on VMIPS for the vector operation $A = B \times s$, where s is a scalar and the length of the vectors A and B is 200?

Answer Assume that the addresses of A and B are initially in R_a and R_b , s is in F_s , and recall that for MIPS (and VMIPS) R_0 always holds 0. Since $(200 \bmod 64) = 8$, the first iteration of the strip-mined loop will execute for a vector length of 8 elements, and the following iterations will execute for a vector length of 64 elements. The starting byte addresses of the next segment of each vector is eight times the vector length. Since the vector length is either 8 or 64, we increment the address registers by $8 \times 8 = 64$ after the first segment and $8 \times 64 = 512$ for later segments. The total number of bytes in the vector is $8 \times 200 = 1600$, and we test for completion by comparing the address of the next vector segment to the initial address plus 1600. Here is the actual code:

```

DADDUI    R2,R0,#1600 ;total # bytes in vector
DADDU     R2,R2,Ra    ;address of the end of A vector
DADDUI    R1,R0,#8   ;loads length of 1st segment
MTC1      VLR,R1    ;load vector length in VLR
DADDUI    R1,R0,#64  ;length in bytes of 1st segment
DADDUI    R3,R0,#64  ;vector length of other segments
Loop: LV      V1,Rb    ;load B
        MULVS.D V2,V1,Fs  ;vector * scalar
        SV      Ra,V2    ;store A
        DADDU    Ra,Ra,R1  ;address of next segment of A
        DADDU    Rb,Rb,R1  ;address of next segment of B
        DADDUI    R1,R0,#512 ;load byte offset next segment
        MTC1      VLR,R3    ;set length to 64 elements
        DSUBU    R4,R2,Ra    ;at the end of A?
        BNEZ    R4,Loop    ;if not, go back

```

The three vector instructions in the loop are dependent and must go into three convoys, hence $T_{chime} = 3$. Let's use our basic formula:

$$T_n = \left[\frac{n}{MVL} \right] \times (T_{loop} + T_{start}) + n \times T_{chime}$$

$$T_{200} = 4 \times (15 + T_{start}) + 200 \times 3$$

$$T_{200} = 60 + (4 \times T_{start}) + 600 = 660 + (4 \times T_{start})$$

The value of T_{start} is the sum of:

- The vector load start-up of 12 clock cycles
- A 7-clock-cycle start-up for the multiply
- A 12-clock-cycle start-up for the store

Thus, the value of T_{start} is given by:

$$T_{start} = 12 + 7 + 12 = 31$$

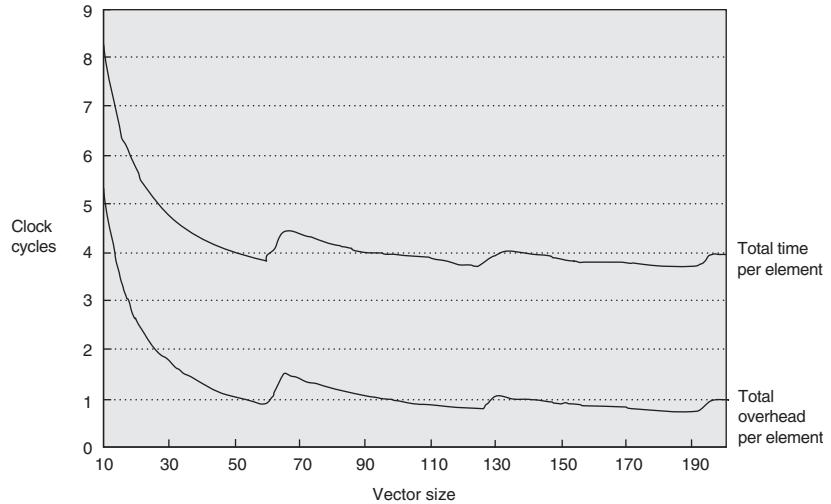


Figure G.5 The total execution time per element and the total overhead time per element versus the vector length for the example on page F-6. For short vectors, the total start-up time is more than one-half of the total time, while for long vectors it reduces to about one-third of the total time. The sudden jumps occur when the vector length crosses a multiple of 64, forcing another iteration of the strip-mining code and execution of a set of vector instructions. These operations increase T_n by $T_{loop} + T_{start}$.

So, the overall value becomes:

$$T_{200} = 660 + 4 \times 31 = 784$$

The execution time per element with all start-up costs is then $784/200 = 3.9$, compared with a chime approximation of three. In Section G.4, we will be more ambitious—allowing overlapping of separate convoys.

Figure G.5 shows the overhead and effective rates per element for the previous example ($A = B \times s$) with various vector lengths. A chime-counting model would lead to 3 clock cycles per element, while the two sources of overhead add 0.9 clock cycles per element in the limit.

Pipelined Instruction Start-Up and Multiple Lanes

Adding multiple lanes increases peak performance but does not change start-up latency, and so it becomes critical to reduce start-up overhead by allowing the start of one vector instruction to be overlapped with the completion of preceding vector instructions. The simplest case to consider is when two vector instructions access a different set of vector registers. For example, in the code sequence

```
ADDV.D V1,V2,V3
ADDV.D V4,V5,V6
```

An implementation can allow the first element of the second vector instruction to follow immediately the last element of the first vector instruction down the FP adder pipeline. To reduce the complexity of control logic, some vector machines require some *recovery time* or *dead time* in between two vector instructions dispatched to the same vector unit. Figure G.6 is a pipeline diagram that shows both start-up latency and dead time for a single vector pipeline.

The following example illustrates the impact of this dead time on achievable vector performance.

Example The Cray C90 has two lanes but requires 4 clock cycles of dead time between any two vector instructions to the same functional unit, even if they have no data dependences. For the maximum vector length of 128 elements, what is the reduction in achievable peak performance caused by the dead time? What would be the reduction if the number of lanes were increased to 16?

Answer A maximum length vector of 128 elements is divided over the two lanes and occupies a vector functional unit for 64 clock cycles. The dead time adds another 4 cycles of occupancy, reducing the peak performance to $64/(64+4) = 94.1\%$ of the value without dead time. If the number of lanes is increased to 16, maximum length vector instructions will occupy a functional unit for only $128/16 = 8$ cycles, and the dead time will reduce peak performance to $8/(8+4) = 66.6\%$ of the value without dead time. In this second case, the vector units can never be more than 2/3 busy!

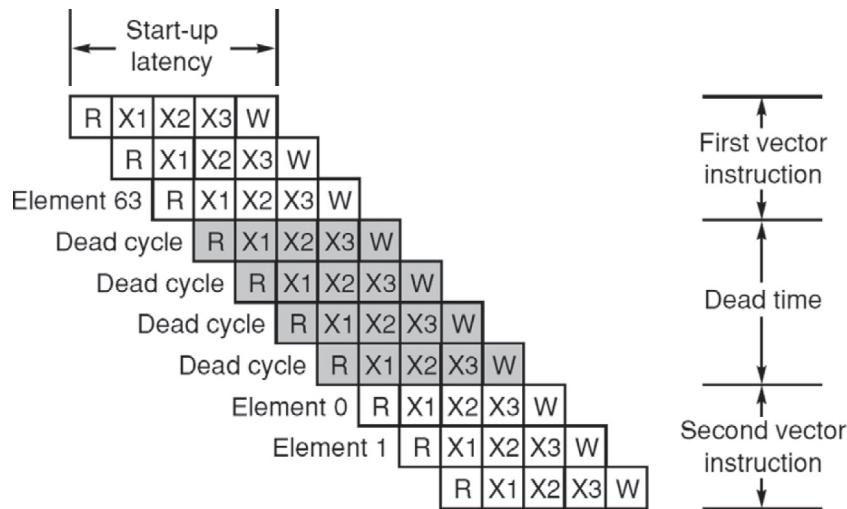


Figure G.6 Start-up latency and dead time for a single vector pipeline. Each element has a 5-cycle latency: 1 cycle to read the vector-register file, 3 cycles in execution, then 1 cycle to write the vector-register file. Elements from the same vector instruction can follow each other down the pipeline, but this machine inserts 4 cycles of dead time between two different vector instructions. The dead time can be eliminated with more complex control logic. (Reproduced with permission from Asanovic [1998].)

Pipelining instruction start-up becomes more complicated when multiple instructions can be reading and writing the same vector register and when some instructions may stall unpredictably—for example, a vector load encountering memory bank conflicts. However, as both the number of lanes and pipeline latencies increase, it becomes increasingly important to allow fully pipelined instruction start-up.

G.3

Vector Memory Systems in More Depth

To maintain an initiation rate of one word fetched or stored per clock, the memory system must be capable of producing or accepting this much data. As we saw in Chapter 4, this usually done by spreading accesses across multiple independent memory banks. Having significant numbers of banks is useful for dealing with vector loads or stores that access rows or columns of data.

The desired access rate and the bank access time determined how many banks were needed to access memory without stalls. This example shows how these timings work out in a vector processor.

Example Suppose we want to fetch a vector of 64 elements starting at byte address 136, and a memory access takes 6 clocks. How many memory banks must we have to support one fetch per clock cycle? With what addresses are the banks accessed? When will the various elements arrive at the CPU?

Answer Six clocks per access require at least 6 banks, but because we want the number of banks to be a power of 2, we choose to have 8 banks. Figure G.7 shows the timing for the first few sets of accesses for an 8-bank system with a 6-clock-cycle access latency.

The timing of real memory banks is usually split into two different components, the access latency and the bank cycle time (or *bank busy time*). The access latency is the time from when the address arrives at the bank until the bank returns a data value, while the busy time is the time the bank is occupied with one request. The access latency adds to the start-up cost of fetching a vector from memory (the total memory latency also includes time to traverse the pipelined interconnection networks that transfer addresses and data between the CPU and memory banks). The bank busy time governs the effective bandwidth of a memory system because a processor cannot issue a second request to the same bank until the bank busy time has elapsed.

For simple unpipelined SRAM banks as used in the previous examples, the access latency and busy time are approximately the same. For a pipelined SRAM bank, however, the access latency is larger than the busy time because each element access only occupies one stage in the memory bank pipeline. For a DRAM bank, the access latency is usually shorter than the busy time because a DRAM needs extra time to restore the read value after the destructive read operation. For memory systems that support multiple simultaneous vector accesses

Cycle no.	Bank							
	0	1	2	3	4	5	6	7
0		136						
1		Busy	144					
2		Busy	Busy	152				
3		Busy	Busy	Busy	160			
4		Busy	Busy	Busy	Busy	168		
5		Busy	Busy	Busy	Busy	Busy	176	
6			Busy	Busy	Busy	Busy	Busy	184
7	192			Busy	Busy	Busy	Busy	Busy
8	Busy	200			Busy	Busy	Busy	Busy
9	Busy	Busy	208			Busy	Busy	Busy
10	Busy	Busy	Busy	216			Busy	Busy
11	Busy	Busy	Busy	Busy	224			Busy
12	Busy	Busy	Busy	Busy	Busy	232		
13		Busy	Busy	Busy	Busy	Busy	240	
14			Busy	Busy	Busy	Busy	Busy	248
15	256			Busy	Busy	Busy	Busy	Busy
16	Busy	264			Busy	Busy	Busy	Busy

Figure G.7 Memory addresses (in bytes) by bank number and time slot at which access begins. Each memory bank latches the element address at the start of an access and is then busy for 6 clock cycles before returning a value to the CPU. Note that the CPU cannot keep all 8 banks busy all the time because it is limited to supplying one new address and receiving one data item each cycle.

or allow nonsequential accesses in vector loads or stores, the number of memory banks should be larger than the minimum; otherwise, memory bank conflicts will exist.

Memory bank conflicts will not occur within a single vector memory instruction if the stride and number of banks are relatively prime with respect to each other and there are enough banks to avoid conflicts in the unit stride case. When there are no bank conflicts, multiword and unit strides run at the same rates. Increasing the number of memory banks to a number greater than the minimum to prevent stalls with a stride of length 1 will decrease the stall frequency for some other strides. For example, with 64 banks, a stride of 32 will stall on every other access, rather than every access. If we originally had a stride of 8 and 16 banks, every other access would stall; with 64 banks, a stride of 8 will stall on every eighth access. If we have multiple memory pipelines and/or multiple processors sharing the same memory system, we will also need more banks to prevent conflicts. Even machines with a single memory pipeline can experience memory bank conflicts on unit stride

accesses between the last few elements of one instruction and the first few elements of the next instruction, and increasing the number of banks will reduce the probability of these inter-instruction conflicts. In 2011, most vector supercomputers spread the accesses from each CPU across hundreds of memory banks. Because bank conflicts can still occur in non-unit stride cases, programmers favor unit stride accesses whenever possible.

A modern supercomputer may have dozens of CPUs, each with multiple memory pipelines connected to thousands of memory banks. It would be impractical to provide a dedicated path between each memory pipeline and each memory bank, so, typically, a multistage switching network is used to connect memory pipelines to memory banks. Congestion can arise in this switching network as different vector accesses contend for the same circuit paths, causing additional stalls in the memory system.

G.4

Enhancing Vector Performance

In this section, we present techniques for improving the performance of a vector processor in more depth than we did in Chapter 4.

Chaining in More Depth

Early implementations of chaining worked like forwarding, but this restricted the timing of the source and destination instructions in the chain. Recent implementations use *flexible chaining*, which allows a vector instruction to chain to essentially any other active vector instruction, assuming that no structural hazard is generated. Flexible chaining requires simultaneous access to the same vector register by different vector instructions, which can be implemented either by adding more read and write ports or by organizing the vector-register file storage into interleaved banks in a similar way to the memory system. We assume this type of chaining throughout the rest of this appendix.

Even though a pair of operations depends on one another, chaining allows the operations to proceed in parallel on separate elements of the vector. This permits the operations to be scheduled in the same convoy and reduces the number of chimes required. For the previous sequence, a sustained rate (ignoring start-up) of two floating-point operations per clock cycle, or one chime, can be achieved, even though the operations are dependent! The total running time for the above sequence becomes:

$$\text{Vector length} + \text{Start-up time}_{\text{ADDV}} + \text{Start-up time}_{\text{MULV}}$$

Figure G.8 shows the timing of a chained and an unchained version of the above pair of vector instructions with a vector length of 64. This convoy requires one chime; however, because it uses chaining, the start-up overhead will be seen in the actual timing of the convoy. In Figure G.8, the total time for chained operation is 77 clock cycles, or 1.2 cycles per result. With 128 floating-point operations done in that time, 1.7 FLOPS per clock cycle are obtained. For the unchained version, there are 141 clock cycles, or 0.9 FLOPS per clock cycle.

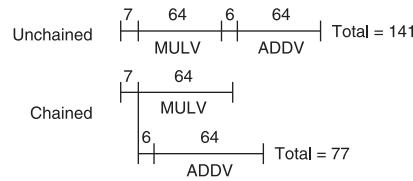


Figure G.8 Timings for a sequence of dependent vector operations ADDV and MULV, both unchained and chained. The 6- and 7-clock-cycle delays are the latency of the adder and multiplier.

Although chaining allows us to reduce the chime component of the execution time by putting two dependent instructions in the same convoy, it does not eliminate the start-up overhead. If we want an accurate running time estimate, we must count the start-up time both within and across convoys. With chaining, the number of chimes for a sequence is determined by the number of different vector functional units available in the processor and the number required by the application. In particular, no convoy can contain a structural hazard. This means, for example, that a sequence containing two vector memory instructions must take at least two convoys, and hence two chimes, on a processor like VMIPS with only one vector load-store unit.

Chaining is so important that every modern vector processor supports flexible chaining.

Sparse Matrices in More Depth

Chapter 4 shows techniques to allow programs with sparse matrices to execute in vector mode. Let's start with a quick review. In a sparse matrix, the elements of a vector are usually stored in some compacted form and then accessed indirectly. Assuming a simplified sparse structure, we might see code that looks like this:

```
do      100 i = 1, n
100          A(K(i)) = A(K(i)) + C(M(i))
```

This code implements a sparse vector sum on the arrays A and C, using index vectors K and M to designate the nonzero elements of A and C. (A and C must have the same number of nonzero elements— n of them.) Another common representation for sparse matrices uses a bit vector to show which elements exist and a dense vector for the nonzero elements. Often both representations exist in the same program. Sparse matrices are found in many codes, and there are many ways to implement them, depending on the data structure used in the program.

A simple vectorizing compiler could not automatically vectorize the source code above because the compiler would not know that the elements of K are distinct values and thus that no dependences exist. Instead, a programmer directive would tell the compiler that it could run the loop in vector mode.

More sophisticated vectorizing compilers can vectorize the loop automatically without programmer annotations by inserting run time checks for data

dependences. These run time checks are implemented with a vectorized software version of the advanced load address table (ALAT) hardware described in Appendix H for the Itanium processor. The associative ALAT hardware is replaced with a software hash table that detects if two element accesses within the same stripmine iteration are to the same address. If no dependences are detected, the stripmine iteration can complete using the maximum vector length. If a dependence is detected, the vector length is reset to a smaller value that avoids all dependency violations, leaving the remaining elements to be handled on the next iteration of the stripmined loop. Although this scheme adds considerable software overhead to the loop, the overhead is mostly vectorized for the common case where there are no dependences; as a result, the loop still runs considerably faster than scalar code (although much slower than if a programmer directive was provided).

A scatter-gather capability is included on many of the recent supercomputers. These operations often run more slowly than strided accesses because they are more complex to implement and are more susceptible to bank conflicts, but they are still much faster than the alternative, which may be a scalar loop. If the sparsity properties of a matrix change, a new index vector must be computed. Many processors provide support for computing the index vector quickly. The CVI (create vector index) instruction in VMIPS creates an index vector given a stride (m), where the values in the index vector are $0, m, 2 \times m, \dots, 63 \times m$. Some processors provide an instruction to create a compressed index vector whose entries correspond to the positions with a one in the mask register. Other vector architectures provide a method to compress a vector. In VMIPS, we define the CVI instruction to always create a compressed index vector using the vector mask. When the vector mask is all ones, a standard index vector will be created.

The indexed loads-stores and the CVI instruction provide an alternative method to support conditional vector execution. Let us first recall code from Chapter 4:

```

low = 1
VL = (n mod MVL) /*find the odd-size piece*/
do 1 j = 0,(n/MVL) /*outer loop*/
    do 10 i = low, low + VL - 1 /*runs for length VL*/
        Y(i) = a * X(i) + Y(i) /*main operation*/
10    continue
    low = low + VL /*start of next vector*/
    VL = MVL /*reset the length to max*/
1    continue

```

Here is a vector sequence that implements that loop using CVI:

LV	V1,Ra	;load vector A into V1
L.D	F0,#0	;load FP zero into F0
SNEVS.D	V1,F0	;sets the VM to 1 if V1(i)!=F0
CVI	V2,#8	;generates indices in V2
POP	R1,VM	;find the number of 1's in VM
MTC1	VLR,R1	;load vector-length register
CVM		;clears the mask

```

LVI      V3,(Ra+V2) ;load the nonzero A elements
LVI      V4,(Rb+V2) ;load corresponding B elements
SUBV.D  V3,V3,V4   ;do the subtract
SVI      (Ra+V2),V3  ;store A back

```

Whether the implementation using scatter-gather is better than the conditionally executed version depends on the frequency with which the condition holds and the cost of the operations. Ignoring chaining, the running time of the original version is $5n + c_1$. The running time of the second version, using indexed loads and stores with a running time of one element per clock, is $4n + 4fn + c_2$, where f is the fraction of elements for which the condition is true (i.e., $A(i) \neq 0$). If we assume that the values of c_1 and c_2 are comparable, or that they are much smaller than n , we can find when this second technique is better.

$$\text{Time}_1 = 5(n)$$

$$\text{Time}_2 = 4n + 4fn$$

We want $\text{Time}_1 > \text{Time}_2$, so

$$\begin{aligned} 5n &> 4n + 4fn \\ \frac{1}{4} &> f \end{aligned}$$

That is, the second method is faster if less than one-quarter of the elements are non-zero. In many cases, the frequency of execution is much lower. If the index vector can be reused, or if the number of vector statements within the if statement grows, the advantage of the scatter-gather approach will increase sharply.

G.5

Effectiveness of Compiler Vectorization

Two factors affect the success with which a program can be run in vector mode. The first factor is the structure of the program itself: Do the loops have true data dependences, or can they be restructured so as not to have such dependences? This factor is influenced by the algorithms chosen and, to some extent, by how they are coded. The second factor is the capability of the compiler. While no compiler can vectorize a loop where no parallelism among the loop iterations exists, there is tremendous variation in the ability of compilers to determine whether a loop can be vectorized. The techniques used to vectorize programs are the same as those discussed in Chapter 3 for uncovering ILP; here, we simply review how well these techniques work.

There is tremendous variation in how well different compilers do in vectorizing programs. As a summary of the state of vectorizing compilers, consider the data in Figure G.9, which shows the extent of vectorization for different processors using a test suite of 100 handwritten FORTRAN kernels. The kernels were designed to test vectorization capability and can all be vectorized by hand; we will see several examples of these loops in the exercises.

Processor	Compiler	Completely vectorized	Partially vectorized	Not vectorized
CDC CYBER 205	VAST-2 V2.21	62	5	33
Convex C-series	FC5.0	69	5	26
Cray X-MP	CFT77 V3.0	69	3	28
Cray X-MP	CFT V1.15	50	1	49
Cray-2	CFT2 V3.1a	27	1	72
ETA-10	FTN 77 V1.0	62	7	31
Hitachi S810/820	FORT77/HAP V20-2B	67	4	29
IBM 3090/VF	VS FORTRAN V2.4	52	4	44
NEC SX/2	FORTRAN77 / SX V.040	66	5	29

Figure G.9 Result of applying vectorizing compilers to the 100 FORTRAN test kernels. For each processor we indicate how many loops were completely vectorized, partially vectorized, and unvectorized. These loops were collected by Callahan, Dongarra, and Levine [1988]. Two different compilers for the Cray X-MP show the large dependence on compiler technology.

G.6

Putting It All Together: Performance of Vector Processors

In this section, we look at performance measures for vector processors and what they tell us about the processors. To determine the performance of a processor on a vector problem we must look at the start-up cost and the sustained rate. The simplest and best way to report the performance of a vector processor on a loop is to give the execution time of the vector loop. For vector loops, people often give the MFLOPS (millions of floating-point operations per second) rating rather than execution time. We use the notation R_n for the MFLOPS rating on a vector of length n . Using the measurements T_n (time) or R_n (rate) is equivalent if the number of FLOPS is agreed upon. In any event, either measurement should include the overhead.

In this section, we examine the performance of VMIPS on a DAXPY loop (see Chapter 4) by looking at performance from different viewpoints. We will continue to compute the execution time of a vector loop using the equation developed in Section G.2. At the same time, we will look at different ways to measure performance using the computed time. The constant values for T_{loop} used in this section introduce some small amount of error, which will be ignored.

Measures of Vector Performance

Because vector length is so important in establishing the performance of a processor, length-related measures are often applied in addition to time and MFLOPS. These length-related measures tend to vary dramatically across different processors

and are interesting to compare. (Remember, though, that *time* is always the measure of interest when comparing the relative speed of two processors.) Three of the most important length-related measures are

- R_∞ —The MFLOPS rate on an infinite-length vector. Although this measure may be of interest when estimating peak performance, real problems have limited vector lengths, and the overhead penalties encountered in real problems will be larger.
- $N_{1/2}$ —The vector length needed to reach one-half of R_∞ . This is a good measure of the impact of overhead.
- N_v —The vector length needed to make vector mode faster than scalar mode. This measures both overhead and the speed of scalars relative to vectors.

Let's look at these measures for our DAXPY problem running on VMIPS. When chained, the inner loop of the DAXPY code in convoys looks like Figure G.10 (assuming that R_x and R_y hold starting addresses).

Recall our performance equation for the execution time of a vector loop with n elements, T_n :

$$T_n = \left[\frac{n}{MVL} \right] \times (T_{\text{loop}} + T_{\text{start}}) + n \times T_{\text{chime}}$$

Chaining allows the loop to run in three chimes (and no less, since there is one memory pipeline); thus, $T_{\text{chime}}=3$. If T_{chime} were a complete indication of performance, the loop would run at an MFLOPS rate of $2/3 \times \text{clock rate}$ (since there are 2 FLOPS per iteration). Thus, based only on the chime count, a 500 MHz VMIPS would run this loop at 333 MFLOPS assuming no strip-mining or start-up overhead. There are several ways to improve the performance: Add additional vector load-store units, allow convoys to overlap to reduce the impact of start-up overheads, and decrease the number of loads required by vector-register allocation. We will examine the first two extensions in this section. The last optimization is actually used for the Cray-1, VMIPS's cousin, to boost the performance by 50%. Reducing the number of loads requires an interprocedural optimization; we examine this transformation in Exercise G.6. Before we examine the first two extensions, let's see what the real performance, including overhead, is.

LV V1,Rx	MULVS.D V2,V1,F0	Convoy 1: chained load and multiply
LV V3,Ry	ADDV.D V4,V2,V3	Convoy 2: second load and add, chained
SV Ry,V4		Convoy 3: store the result

Figure G.10 The inner loop of the DAXPY code in chained convoys.

The Peak Performance of VMIPS on DAXPY

First, we should determine what the peak performance, R_∞ , really is, since we know it must differ from the ideal 333 MFLOPS rate. For now, we continue to use the simplifying assumption that a convoy cannot start until all the instructions in an earlier convoy have completed; later we will remove this restriction. Using this simplification, the start-up overhead for the vector sequence is simply the sum of the start-up times of the instructions:

$$T_{\text{start}} = 12 + 7 + 12 + 6 + 12 = 49$$

Using $MVL=64$, $T_{\text{loop}}=15$, $T_{\text{start}}=49$, and $T_{\text{chime}}=3$ in the performance equation, and assuming that n is not an exact multiple of 64, the time for an n -element operation is

$$\begin{aligned} T_n &= \left[\frac{n}{64} \right] \times (15 + 49) + 3n \\ &\leq (n + 64) + 3n \\ &= 4n + 64 \end{aligned}$$

The sustained rate is actually over 4 clock cycles per iteration, rather than the theoretical rate of 3 chimes, which ignores overhead. The major part of the difference is the cost of the start-up overhead for each block of 64 elements (49 cycles versus 15 for the loop overhead).

We can now compute R_∞ for a 500 MHz clock as:

$$R_\infty = \lim_{n \rightarrow \infty} \left(\frac{\text{Operations per iteration} \times \text{Clock rate}}{\text{Clock cycles per iteration}} \right)$$

The numerator is independent of n , hence

$$R_\infty = \frac{\text{Operations per iteration} \times \text{Clock rate}}{\lim_{n \rightarrow \infty} (\text{Clock cycles per iteration})}$$

$$\lim_{n \rightarrow \infty} (\text{Clock cycles per iteration}) = \lim_{n \rightarrow \infty} \left(\frac{T_n}{n} \right) = \lim_{n \rightarrow \infty} \left(\frac{4n + 64}{n} \right) = 4$$

$$R_\infty = \frac{2 \times 500 \text{ MHz}}{4} = 250 \text{ MFLOPS}$$

The performance without the start-up overhead, which is the peak performance given the vector functional unit structure, is now 1.33 times higher. In actuality, the gap between peak and sustained performance for this benchmark is even larger!

Sustained Performance of VMIPS on the Linpack Benchmark

The Linpack benchmark is a Gaussian elimination on a 100×100 matrix. Thus, the vector element lengths range from 99 down to 1. A vector of length k is used k times. Thus, the average vector length is given by:

$$\frac{\sum_{i=1}^{99} i^2}{\sum_{i=1}^{99} i} = 66.3$$

Now we can obtain an accurate estimate of the performance of DAXPY using a vector length of 66:

$$T_{66} = 2 \times (15 + 49) + 66 \times 3 = 128 + 198 = 326$$

$$R_{66} = \frac{2 \times 66 \times 500}{326} \text{ MFLOPS} = 202 \text{ MFLOPS}$$

The peak number, ignoring start-up overhead, is 1.64 times higher than this estimate of sustained performance on the real vector lengths. In actual practice, the Linpack benchmark contains a nontrivial fraction of code that cannot be vectorized. Although this code accounts for less than 20% of the time before vectorization, it runs at less than one-tenth of the performance when counted as FLOPS. Thus, Amdahl's law tells us that the overall performance will be significantly lower than the performance estimated from analyzing the inner loop.

Since vector length has a significant impact on performance, the $N_{1/2}$ and N_v measures are often used in comparing vector machines.

Example What is $N_{1/2}$ for just the inner loop of DAXPY for VMIPS with a 500 MHz clock?

Answer Using R_∞ as the peak rate, we want to know the vector length that will achieve about 125 MFLOPS. We start with the formula for MFLOPS assuming that the measurement is made for $N_{1/2}$ elements:

$$\text{MFLOPS} = \frac{\text{FLOPS executed in } N_{1/2} \text{ iterations}}{\text{Clock cycles to execute } N_{1/2} \text{ iterations}} \times \frac{\text{Clock cycles}}{\text{Second}} \times 10^{-6}$$

$$125 = \frac{2 \times N_{1/2}}{T_{N_{1/2}}} \times 500$$

Simplifying this and then assuming $N_{1/2} < 64$, so that $T_{N_{1/2} < 64} = 64 + 3 \times n$, yields:

$$T_{N_{1/2}} = 8 \times N_{1/2}$$

$$64 + 3 \times N_{1/2} = 8 \times N_{1/2}$$

$$5 \times N_{1/2} = 64$$

$$N_{1/2} = 12.8$$

So $N_{1/2} = 13$; that is, a vector of length 13 gives approximately one-half the peak performance for the DAXPY loop on VMIPS.

Example What is the vector length, N_v , such that the vector operation runs faster than the scalar?

Answer Again, we know that $N_v < 64$. The time to do one iteration in scalar mode can be estimated as $10 + 12 + 12 + 7 + 6 + 12 = 59$ clocks, where 10 is the estimate of the loop overhead, known to be somewhat less than the strip-mining loop overhead. In the last problem, we showed that this vector loop runs in vector mode in time $T_{n \leq 64} = 64 + 3 \times n$ clock cycles. Therefore,

$$64 + 3N_v = 59N_v$$

$$N_v = \left[\frac{64}{56} \right]$$

$$N_v = 2$$

For the DAXPY loop, vector mode is faster than scalar as long as the vector has at least two elements. This number is surprisingly small.

DAXPY Performance on an Enhanced VMIPS

DAXPY, like many vector problems, is memory limited. Consequently, performance could be improved by adding more memory access pipelines. This is the major architectural difference between the Cray X-MP (and later processors) and the Cray-1. The Cray X-MP has three memory pipelines, compared with the Cray-1's single memory pipeline, and the X-MP has more flexible chaining. How does this affect performance?

Example What would be the value of T_{66} for DAXPY on VMIPS if we added two more memory pipelines?

Answer With three memory pipelines, all the instructions fit in one convoy and take one chime. The start-up overheads are the same, so

$$T_{66} = \left[\frac{66}{64} \right] \times (T_{\text{loop}} + T_{\text{start}}) + 66 \times T_{\text{chime}}$$

$$T_{66} = 2 \times (15 + 49) + 66 \times 1 = 194$$

With three memory pipelines, we have reduced the clock cycle count for sustained performance from 326 to 194, a factor of 1.7. Note the effect of Amdahl's law: We improved the theoretical peak rate as measured by the number of chimes by a factor of 3, but only achieved an overall improvement of a factor of 1.7 in sustained performance.

Another improvement could come from allowing different convoys to overlap and also allowing the scalar loop overhead to overlap with the vector instructions. This requires that one vector operation be allowed to begin using a functional unit before another operation has completed, which complicates the instruction issue logic. Allowing this overlap eliminates the separate start-up overhead for every convoy except the first and hides the loop overhead as well.

To achieve the maximum hiding of strip-mining overhead, we need to be able to overlap strip-mined instances of the loop, allowing two instances of a convoy as well as possibly two instances of the scalar code to be in execution simultaneously. This requires the same techniques we looked at in Chapter 3 to avoid WAR hazards, although because no overlapped read and write of a single vector element is possible, copying can be avoided. This technique, called *tailgating*, was used in the Cray-2. Alternatively, we could unroll the outer loop to create several instances of the vector sequence using different register sets (assuming sufficient registers), just as we did in Chapter 3. By allowing maximum overlap of the convoys and the scalar loop overhead, the start-up and loop overheads will only be seen *once* per vector sequence, independent of the number of convoys and the instructions in each convoy. In this way, a processor with vector registers can have both low start-up overhead for short vectors and high peak performance for very long vectors.

Example What would be the values of R_∞ and T_{66} for DAXPY on VMIPS if we added two more memory pipelines and allowed the strip-mining and start-up overheads to be fully overlapped?

Answer

$$R_\infty = \lim_{n \rightarrow \infty} \left(\frac{\text{Operations per iteration} \times \text{Clock rate}}{\text{Clock cycles per iteration}} \right)$$

$$\lim_{n \rightarrow \infty} (\text{Clock cycles per iteration}) = \lim_{n \rightarrow \infty} \left(\frac{T_n}{n} \right)$$

Since the overhead is only seen once, $T_n = n + 49 + 15 = n + 64$. Thus,

$$\lim_{n \rightarrow \infty} \left(\frac{T_n}{n} \right) = \lim_{n \rightarrow \infty} \left(\frac{n + 64}{n} \right) = 1$$

$$R_\infty = \frac{2 \times 500 \text{ MHz}}{1} = 1000 \text{ MFLOPS}$$

Adding the extra memory pipelines and more flexible issue logic yields an improvement in peak performance of a factor of 4. However, $T_{66} = 130$, so for shorter vectors the sustained performance improvement is about $326/130 = 2.5$ times.

In summary, we have examined several measures of vector performance. Theoretical peak performance can be calculated based purely on the value of T_{chime} as:

$$\frac{\text{Number of FLOPS per iteration} \times \text{Clock rate}}{T_{\text{chime}}}$$

By including the loop overhead, we can calculate values for peak performance for an infinite-length vector (R_{∞}) and also for sustained performance, R_n for a vector of length n , which is computed as:

$$R_n = \frac{\text{Number of FLOPS per iteration} \times n \times \text{Clock rate}}{T_n}$$

Using these measures we also can find $N_{1/2}$ and N_v , which give us another way of looking at the start-up overhead for vectors and the ratio of vector to scalar speed. A wide variety of measures of performance of vector processors is useful in understanding the range of performance that applications may see on a vector processor.

G.7

A Modern Vector Supercomputer: The Cray X1

The Cray X1 was introduced in 2002, and, together with the NEC SX/8, represents the state of the art in modern vector supercomputers. The X1 system architecture supports thousands of powerful vector processors sharing a single global memory.

The Cray X1 has an unusual processor architecture, shown in Figure G.11. A large Multi-Streaming Processor (MSP) is formed by ganging together four Single-Streaming Processors (SSPs). Each SSP is a complete single-chip vector microprocessor, containing a scalar unit, scalar caches, and a two-lane vector unit. The SSP scalar unit is a dual-issue out-of-order superscalar processor with a 16 KB instruction cache and a 16 KB scalar write-through data cache, both two-way set associative with 32-byte cache lines. The SSP vector unit contains a vector register file, three vector arithmetic units, and one vector load-store unit. It is much easier to pipeline deeply a vector functional unit than a superscalar issue mechanism, so the X1 vector unit runs at twice the clock rate (800 MHz) of the scalar unit (400 MHz). Each lane can perform a 64-bit floating-point add and a 64-bit floating-point multiply each cycle, leading to a peak performance of 12.8 GFLOPS per MSP.

All previous Cray machines could trace their instruction set architecture (ISA) lineage back to the original Cray-1 design from 1976, with 8 primary registers each for addresses, scalar data, and vector data. For the X1, the ISA was redesigned from scratch to incorporate lessons learned over the last 30 years of compiler and micro-architecture research. The X1 ISA includes 64 64-bit scalar address registers and 64 64-bit scalar data registers, with 32 vector data registers (64 bits per element) and 8 vector mask registers (1 bit per element). The large increase in the number of registers allows the compiler to map more program variables into registers to reduce memory traffic and also allows better static scheduling of code to improve

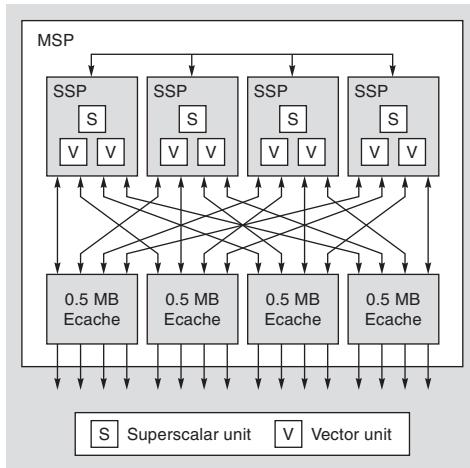


Figure G.11 Cray MSP module. (From Dunnigan et al. [2005].)

run time overlap of instruction execution. Earlier Crays had a compact variable-length instruction set, but the X1 ISA has fixedlength instructions to simplify superscalar fetch and decode.

Four SSP chips are packaged on a multichip module together with four cache chips implementing an external 2 MB cache (Ecache) shared by all the SSPs. The Ecache is two-way set associative with 32-byte lines and a write-back policy. The Ecache can be used to cache vectors, reducing memory traffic for codes that exhibit temporal locality. The ISA also provides vector load and store instruction variants that do not allocate in cache to avoid polluting the Ecache with data that is known to have low locality. The Ecache has sufficient bandwidth to supply one 64-bit word per lane per 800 MHz clock cycle, or over 50 GB/sec per MSP.

At the next level of the X1 packaging hierarchy, shown in Figure G.12, four MSPs are placed on a single printed circuit board together with 16 memory controller chips and DRAM to form an X1 node. Each memory controller chip has eight separate Rambus DRAM channels, where each channel provides 1.6 GB/sec of memory bandwidth. Across all 128 memory channels, the node has over 200 GB/sec of main memory bandwidth.

An X1 system can contain up to 1024 nodes (4096 MSPs or 16,384 SSPs), connected via a very high-bandwidth global network. The network connections are made via the memory controller chips, and all memory in the system is directly accessible from any processor using load and store instructions. This provides much faster global communication than the message-passing protocols used in cluster-based systems. Maintaining cache coherence across such a large number of high-bandwidth shared-memory nodes would be challenging. The approach taken in the X1 is to restrict each Ecache to cache data only from the local node DRAM. The memory controllers implement a directory scheme to maintain

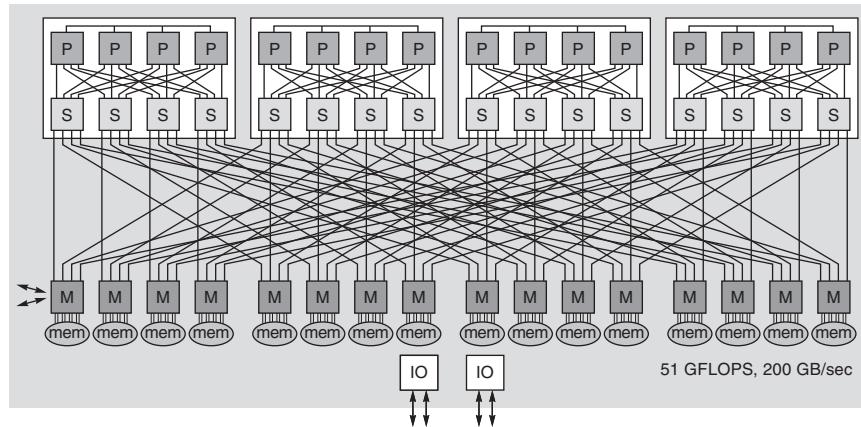


Figure G.12 Cray X1 node. (From Tanqueray [2002].)

coherency between the four Ecaches on a node. Accesses from remote nodes will obtain the most recent version of a location, and remote stores will invalidate local Ecaches before updating memory, but the remote node cannot cache these local locations.

Vector loads and stores are particularly useful in the presence of long-latency cache misses and global communications, as relatively simple vector hardware can generate and track a large number of in-flight memory requests. Contemporary superscalar microprocessors support only 8 to 16 outstanding cache misses, whereas each MSP processor can have up to 2048 outstanding memory requests (512 per SSP). To compensate, superscalar microprocessors have been moving to larger cache line sizes (128 bytes and above) to bring in more data with each cache miss, but this leads to significant wasted bandwidth on non-unit stride accesses over large datasets. The X1 design uses short 32-byte lines throughout to reduce bandwidth waste and instead relies on supporting many independent cache misses to sustain memory bandwidth. This latency tolerance together with the huge memory bandwidth for non-unit strides explains why vector machines can provide large speedups over superscalar microprocessors for certain codes.

Multi-Streaming Processors

The Multi-Streaming concept was first introduced by Cray in the SV1, but has been considerably enhanced in the X1. The four SSPs within an MSP share Ecache, and there is hardware support for barrier synchronization across the four SSPs within an MSP. Each X1 SSP has a two-lane vector unit with 32 vector registers each holding 64 elements. The compiler has several choices as to how to use the SSPs within an MSP.

The simplest use is to gang together four two-lane SSPs to emulate a single eight-lane vector processor. The X1 provides efficient barrier synchronization primitives between SSPs on a node, and the compiler is responsible for generating the MSP code. For example, for a vectorizable inner loop over 1000 elements, the compiler will allocate iterations 0–249 to SSP0, iterations 250–499 to SSP1, iterations 500–749 to SSP2, and iterations 750–999 to SSP3. Each SSP can process its loop iterations independently but must synchronize back with the other SSPs before moving to the next loop nest.

If inner loops do not have many iterations, the eight-lane MSP will have low efficiency, as each SSP will have only a few elements to process and execution time will be dominated by start-up time and synchronization overheads. Another way to use an MSP is for the compiler to parallelize across an outer loop, giving each SSP a different inner loop to process. For example, the following nested loops scale the upper triangle of a matrix by a constant:

```
/* Scale upper triangle by constant K. */
for (row = 0; row < MAX_ROWS; row++)
    for (col = row; col < MAX_COLS; col++)
        A[row][col] = A[row][col] * K;
```

Consider the case where MAX_ROWS and MAX_COLS are both 100 elements. The vector length of the inner loop steps down from 100 to 1 over the iterations of the outer loop. Even for the first inner loop, the loop length would be much less than the maximum vector length (256) of an eight-lane MSP, and the code would therefore be inefficient. Alternatively, the compiler can assign entire inner loops to a single SSP. For example, SSP0 might process rows 0, 4, 8, and so on, while SSP1 processes rows 1, 5, 9, and so on. Each SSP now sees a longer vector. In effect, this approach parallelizes the scalar overhead and makes use of the individual scalar units within each SSP.

Most application code uses MSPs, but it is also possible to compile code to use all the SSPs as individual processors where there is limited vector parallelism but significant thread-level parallelism.

Cray X1E

In 2004, Cray announced an upgrade to the original Cray X1 design. The X1E uses newer fabrication technology that allows two SSPs to be placed on a single chip, making the X1E the first multicore vector microprocessor. Each physical node now contains eight MSPs, but these are organized as two logical nodes of four MSPs each to retain the same programming model as the X1. In addition, the clock rates were raised from 400 MHz scalar and 800 MHz vector to 565 MHz scalar and 1130 MHz vector, giving an improved peak performance of 18 GFLOPS.

G.8**Concluding Remarks**

During the 1980s and 1990s, rapid performance increases in pipelined scalar processors led to a dramatic closing of the gap between traditional vector supercomputers and fast, pipelined, superscalar VLSI microprocessors. In 2011, it is possible to buy a laptop computer for under \$1000 that has a higher CPU clock rate than any available vector supercomputer, even those costing tens of millions of dollars. Although the vector supercomputers have lower clock rates, they support greater parallelism using multiple lanes (up to 16 in the Japanese designs) versus the limited multiple issue of the superscalar microprocessors. Nevertheless, the peak floating-point performance of the low-cost microprocessors is within a factor of two of the leading vector supercomputer CPUs. Of course, high clock rates and high peak performance do not necessarily translate into sustained application performance. Main memory bandwidth is the key distinguishing feature between vector supercomputers and superscalar microprocessor systems.

Providing this large non-unit stride memory bandwidth is one of the major expenses in a vector supercomputer, and traditionally SRAM was used as main memory to reduce the number of memory banks needed and to reduce vector start-up penalties. While SRAM has an access time several times lower than that of DRAM, it costs roughly 10 times as much per bit! To reduce main memory costs and to allow larger capacities, all modern vector supercomputers now use DRAM for main memory, taking advantage of new higher-bandwidth DRAM interfaces such as synchronous DRAM.

This adoption of DRAM for main memory (pioneered by Seymour Cray in the Cray-2) is one example of how vector supercomputers have adapted commodity technology to improve their price-performance. Another example is that vector supercomputers are now including vector data caches. Caches are not effective for all vector codes, however, so these vector caches are designed to allow high main memory bandwidth even in the presence of many cache misses. For example, the Cray X1 MSP can have 2048 outstanding memory loads; for microprocessors, 8 to 16 outstanding cache misses per CPU are more typical maximum numbers.

Another example is the demise of bipolar ECL or gallium arsenide as technologies of choice for supercomputer CPU logic. Because of the huge investment in CMOS technology made possible by the success of the desktop computer, CMOS now offers competitive transistor performance with much greater transistor density and much reduced power dissipation compared with these more exotic technologies. As a result, all leading vector supercomputers are now built with the same CMOS technology as superscalar microprocessors. The primary reason why vector supercomputers have lower clock rates than commodity microprocessors is that they are developed using standard cell ASIC techniques rather than full custom circuit design to reduce the engineering design cost. While a microprocessor design may sell tens of millions of copies and can amortize the design cost over this large number of units, a vector supercomputer is considered a success if over a hundred units are sold!

Conversely, via superscalar microprocessor designs have begun to absorb some of the techniques made popular in earlier vector computer systems, such as with the Multimedia SIMD extensions. As we showed in Chapter 4, the investment in hardware for SIMD performance is increasing rapidly, perhaps even more than for multiprocessors. If the even wider SIMD units of GPUs become well integrated with the scalar cores, including scatter-gather support, we may well conclude that vector architectures have won the architecture wars!

G.9

Historical Perspective and References

This historical perspective adds some details and references that were left out of the version in Chapter 4.

The CDC STAR processor and its descendant, the CYBER 205, were memory-memory vector processors. To keep the hardware simple and support the high bandwidth requirements (up to three memory references per floating-point operation), these processors did not efficiently handle non-unit stride. While most loops have unit stride, a non-unit stride loop had poor performance on these processors because memory-to-memory data movements were required to gather together (and scatter back) the nonadjacent vector elements; these operations used special scatter-gather instructions. In addition, there was special support for sparse vectors that used a bit vector to represent the zeros and nonzeros and a dense vector of nonzero values. These more complex vector operations were slow because of the long memory latency, and it was often faster to use scalar mode for sparse or non-unit stride operations. Schneck [1987] described several of the early pipelined processors (e.g., Stretch) through the first vector processors, including the 205 and Cray-1. Dongarra [1986] did another good survey, focusing on more recent processors.

The 1980s also saw the arrival of smaller-scale vector processors, called mini-supercomputers. Priced at roughly one-tenth the cost of a supercomputer (\$0.5 to \$1 million versus \$5 to \$10 million), these processors caught on quickly. Although many companies joined the market, the two companies that were most successful were Convex and Alliant. Convex started with the uniprocessor C-1 vector processor and then offered a series of small multiprocessors, ending with the C-4 announced in 1994. The keys to the success of Convex over this period were their emphasis on Cray software capability, the effectiveness of their compiler (see Figure G.9), and the quality of their UNIX OS implementation. The C-4 was the last vector machine Convex sold; they switched to making large-scale multiprocessors using Hewlett-Packard RISC microprocessors and were bought by HP in 1995. Alliant [1987] concentrated more on the multiprocessor aspects; they built an eight-processor computer, with each processor offering vector capability. Alliant ceased operation in the early 1990s.

In the early 1980s, CDC spun out a group, called ETA, to build a new supercomputer, the ETA-10, capable of 10 GFLOPS. The ETA processor was delivered in the late 1980s (see Fazio [1987]) and used low-temperature CMOS in a

configuration with up to 10 processors. Each processor retained the memory-memory architecture based on the CYBER 205. Although the ETA-10 achieved enormous peak performance, its scalar speed was not comparable. In 1989, CDC, the first supercomputer vendor, closed ETA and left the supercomputer design business.

In 1986, IBM introduced the System/370 vector architecture (see Moore et al. [1987]) and its first implementation in the 3090 Vector Facility. The architecture extended the System/370 architecture with 171 vector instructions. The 3090/VF was integrated into the 3090 CPU. Unlike most other vector processors of the time, the 3090/VF routed its vectors through the cache. The IBM 370 machines continued to evolve over time and are now called the IBM zSeries. The vector extensions have been removed from the architecture and some of the opcode space was reused to implement 64-bit address extensions.

In late 1989, Cray Research was split into two companies, both aimed at building high-end processors available in the early 1990s. Seymour Cray headed the spin-off, Cray Computer Corporation, until its demise in 1995. Their initial processor, the Cray-3, was to be implemented in gallium arsenide, but they were unable to develop a reliable and cost-effective implementation technology. A single Cray-3 prototype was delivered to the National Center for Atmospheric Research (NCAR) for evaluation purposes in 1993, but no paying customers were found for the design. The Cray-4 prototype, which was to have been the first processor to run at 1 GHz, was close to completion when the company filed for bankruptcy. Shortly before his tragic death in a car accident in 1996, Seymour Cray started yet another company, SRC Computers, to develop high-performance systems but this time using commodity components. In 2000, SRC announced the SRC-6 system, which combined 512 Intel microprocessors, 5 billion gates of reconfigurable logic, and a high-performance vector-style memory system.

Cray Research focused on the C90, a new high-end processor with up to 16 processors and a clock rate of 240 MHz. This processor was delivered in 1991. The J90 was a CMOS-based vector machine using DRAM memory starting at \$250,000, but with typical configurations running about \$1 million. In mid-1995, Cray Research was acquired by Silicon Graphics, and in 1998 released the SV1 system, which grafted considerably faster CMOS processors onto the J90 memory system, and which also added a data cache for vectors to each CPU to help meet the increased memory bandwidth demands. The SV1 also introduced the MSP concept, which was developed to provide competitive single-CPU performance by ganging together multiple slower CPUs. Silicon Graphics sold Cray Research to Tera Computer in 2000, and the joint company was renamed Cray Inc.

The basis for modern vectorizing compiler technology and the notion of data dependence was developed by Kuck and his colleagues [1974] at the University of Illinois. Banerjee [1979] developed the test named after him. Padua and Wolfe [1986] gave a good overview of vectorizing compiler technology.

Benchmark studies of various supercomputers, including attempts to understand the performance differences, have been undertaken by Lubeck, Moore,

and Mendez [1985], Bucher [1983], and Jordan [1987]. There are several benchmark suites aimed at scientific usage and often employed for supercomputer benchmarking, including Linpack and the Lawrence Livermore Laboratories FORTRAN kernels. The University of Illinois coordinated the collection of a set of benchmarks for supercomputers, called the Perfect Club. In 1993, the Perfect Club was integrated into SPEC, which released a set of benchmarks, SPEChpc96, aimed at high-end scientific processing in 1996. The NAS parallel benchmarks developed at the NASA Ames Research Center [Bailey et al. 1991] have become a popular set of kernels and applications used for supercomputer evaluation. A new benchmark suite, HPC Challenge, was introduced consisting of a few kernels that stress machine memory and interconnect bandwidths in addition to floating-point performance [Luszczek et al. 2005]. Although standard supercomputer benchmarks are useful as a rough measure of machine capabilities, large supercomputer purchases are generally preceded by a careful performance evaluation on the actual mix of applications required at the customer site.

References

- Alliant Computer Systems Corp, 1987. Alliant FX/Series: Product Summary. Mass, Acton (June).
- Asanovic, K., 1998. Vector microprocessors," Ph.D. thesis, Computer Science Division. University of California at Berkeley (May).
- Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K., 1991. The NAS parallel benchmarks. *Int'l. J. Supercomputing Applications* 5, 63–73.
- Banerjee, U., 1979. Speedup of ordinary programs," Ph.D. thesis, Department of Computer Science. University of Illinois at Urbana-Champaign (October).
- Baskett, F., Keller, T.W., 1977. An Evaluation of the Cray-1 Processor. In: Kuck, D.J., Lawrie, D.H., Sameh, A.H. (Eds.), *High Speed Computer and Algorithm Organization*. Academic Press, San Diego, pp. 71–84.
- Brandt, M., Brooks, J., Cahir, M., Hewitt, T., Lopez-Pineda, E., Sandness, D., 2000. The Benchmarkers Guide for Cray SV1 Systems. Cray Inc., Seattle, Wash.
- Bucher, I.Y., 1983. The computational speed of supercomputers. In: Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems, August 29–31, 1983. Minneapolis, Minn, pp. 151–165.
- Callahan, D., Dongarra, J., Levine, D., 1988. Vectorizing compilers: A test suite and results. In: *Supercomputing '88: Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, November 12–17, pp. 98–105. Orlando, FL.
- Chen, S., 1983. Large-scale and high-speed multiprocessor system for scientific applications. In: Hwang, K. (Ed.), *Superprocessors: Design and applications*. Proc. NATO Advanced Research Workshop on High-Speed Computing, June 20–22, 1983, Julich, Kernforschungsanlage, Federal Republic of Germany. IEEE, (August), 1984.
- Dongarra, J.J., 1986. A survey of high performance processors. *COMPON*, IEEE, 8–11 (March).
- Dunnigan, T.H., Vetter, J.S., White III, J.B., Worley, P.H., 2005. Performance evaluation of the Cray X1 distributed shared-memory architecture. *IEEE Micro* 25 (1 (January–February)), 30–40.
- Fazio, D., 1987. It's really much more fun building a supercomputer than it is simply inventing one. *COMPON*, IEEE, 102–105 (February).
- Flynn, M.J., 1966. Very high-speed computing systems. In: Proc. IEEE 54:12 (December), pp. 1901–1909.
- Hintz, R.G., Tate, D.P., 1972. Control data STAR-100 processor design. *COMPON*, IEEE 1–4 (September).
- Jordan, K.E., 1987. Performance comparison of large-scale scientific processors: Scalar mainframes, mainframes with vector facilities, and supercomputers. *Computer* 20 (3 (March)), 10–23.

- Kitagawa, K., Tagaya, S., Hagihara, Y., Kanoh, Y., 2003. A hardware overview of SX-6 and SX-7 supercomputer. *NEC Research & Development* J 44 (1 (January)), 2–7.
- Kuck, D., Budnik, P.P., Chen, S.-C., Lawrie, D.H., Towle, R.A., Strebendt, R.E., Davis Jr., E.W., Han, J., Kraska, P.W., Muraoka, Y., 1974. Measurements of parallelism in ordinary FORTRAN programs. *Computer* 7 (1 (January)), 37–46.
- Lincoln, N.R., 1982. Technology and design trade offs in the creation of a modern supercomputer. *IEEE Trans. on Computers*, 363–376. C-31:5 (May).
- Lubeck, O., Moore, J., Mendez, R., 1985. A benchmark comparison of three supercomputers: Fujitsu VP-200, Hitachi S810/20, and Cray X-MP/2. *Computer* 18 (1 (January)), 10–29.
- Luszczek, P., Dongarra, J.J., Koester, D., Rabenseifner, R., Lucas, B., Kepner, J., McCalpin, J., Bailey, D., Takahashi, D., 2005. In: Introduction to the HPC challenge benchmark suite,” Lawrence Berkeley National Laboratory, Paper LBNL-57493 (April 25). <http://repositories.cdlib.org/lbnl/LBNL-57493>.
- Miranker, G.S., Rubenstein, J., Sanguinetti, J., 1988. Squeezing a Cray-class supercomputer into a single-user package. *COMPCON*, IEEE, 452–456 (March).
- Miura, K., Uchida, K., 1983. FACOM vector processing system: VP100/200. In: Proc. NATO Advanced Research Workshop on High-Speed Computing. June 20–22, 1983, Jülich, Kernforschungsanlage, Federal Republic of Germany; also in K. Hwang, ed., “Superprocessors: Design and applications,” *IEEE* (August), 1984, 59–73.
- Moore, B., Padegs, A., Smith, R., Bucholz, W., 1987. Concepts of the System/370 vector architecture. In: Proc. 14th Int'l. Symposium on Computer Architecture, June 3–6, 1987. Pittsburgh, Penn, pp. 282–292.
- Padua, D., Wolfe, M., 1986. Advanced compiler optimizations for supercomputers. *Comm. ACM* 29 (12 (December)), 1184–1201.
- Russell, R.M., 1978. The Cray-1 processor system. *Comm. of the ACM* 21 (1 (January)), 63–72.
- Schneck, P.B., 1987. Superprocessor Architecture. Kluwer Academic Publishers, Norwell, Mass.
- Smith, B.J., 1981. Architecture and applications of the HEP multiprocessor system. *Real-Time Signal Processing IV* 298, 241–248. August.
- Sporer, M., Moss, F.H., Mathais, C.J., 1988. An introduction to the architecture of the Stellar Graphics supercomputer. *COMPON*, IEEE 464 (March).
- Tanqueray, D., 2002. The Cray X1 and supercomputer road map. In: Proc. 13th Daresbury Machine Evaluation Workshop, December 11–12. Cheshire, England.
- Vajapeyam, S., 1991. Instruction-level characterization of the Cray Y-MP processor. Ph.D. thesis, Computer Sciences Department. University of Wisconsin-Madison.
- Watanabe, T., 1987. Architecture and performance of the NEC supercomputer SX system. *Parallel Computing* 5, 247–255.
- Watson, W.J., 1972. The TI ASC—a highly modular and flexible super processor architecture. In: Proc. AFIPS Fall Joint Computer Conf, pp. 221–228.

Exercises

In these exercises assume VMIPS has a clock rate of 500 MHz and that $T_{loop} = 15$. Use the start-up times from Figure G.2, and assume that the store latency is always included in the running time.

- G.1 [10] <G.1, G.2> Write a VMIPS vector sequence that achieves the peak MFLOPS performance of the processor (use the functional unit and instruction description in Section G.2). Assuming a 500-MHz clock rate, what is the peak MFLOPS?
- G.2 [20/15/15] <G.1–G.6> Consider the following vector code run on a 500 MHz version of VMIPS for a fixed vector length of 64:

```

LV      V1,Ra
MULV.D V2,V1,V3
ADDV.D V4,V1,V3
SV      Rb,V2
SV      Rc,V4

```

Ignore all strip-mining overhead, but assume that the store latency must be included in the time to perform the loop. The entire sequence produces 64 results.

- a. [20]<G.1–G.4> Assuming no chaining and a single memory pipeline, how many chimes are required? How many clock cycles per result (including both stores as one result) does this vector sequence require, including start-up overhead?

- b. [15]<G.1–G.4> If the vector sequence is chained, how many clock cycles per result does this sequence require, including overhead?

- c. [15]<G.1–G.6> Suppose VMIPS had three memory pipelines and chaining.

If there were no bank conflicts in the accesses for the above loop, how many clock cycles are required per result for this sequence?

- G.3 [20/20/15/15/20/20/20]<G.2–G.6> Consider the following FORTRAN code:

```
do 10 i=1,n
      A(i)=A(i)+B(i)
      B(i)=x * B(i)
10      continue
```

Use the techniques of Section G.6 to estimate performance throughout this exercise, assuming a 500 MHz version of VMIPS.

- a. [20]<G.2–G.6> Write the best VMIPS vector code for the inner portion of the loop. Assume x is in F0 and the addresses of A and B are in Ra and Rb, respectively.

- b. [20]<G.2–G.6> Find the total time for this loop on VMIPS (T_{100}). What is the MFLOPS rating for the loop (R_{100})?

- c. [15]<G.2–G.6> Find R_∞ for this loop.

- d. [15]<G.2–G.6> Find $N_{1/2}$ for this loop.

- e. [20]<G.2–G.6> Find N_v for this loop. Assume the scalar code has been pipeline scheduled so that each memory reference takes six cycles and each FP operation takes three cycles. Assume the scalar overhead is also T_{loop} .

- f. [20]<G.2–G.6> Assume VMIPS has two memory pipelines. Write vector code that takes advantage of the second memory pipeline. Show the layout in convoys.

- g. [20]<G.2–G.6> Compute T_{100} and R_{100} for VMIPS with two memory pipelines.

- G.4 [20/10]<G.2> Suppose we have a version of VMIPS with eight memory banks (each a double word wide) and a memory access time of eight cycles.

- a. [20]<G.2> If a load vector of length 64 is executed with a stride of 20 double words, how many cycles will the load take to complete?

- b. [10]<G.2> What percentage of the memory bandwidth do you achieve on a 64-element load at stride 20 versus stride 1?

G.5 [12/12]< G.5–G.6 > Consider the following loop:

```
C=0.0
do 10 i=1,64
      A(i)=A(i)+B(i)
      C=C+A(i)
10      continue
```

- a. [12]< G.5–G.6 > Split the loop into two loops: one with no dependence and one with a dependence. Write these loops in FORTRAN—as a source-to-source transformation. This optimization is called *loop fission*.
- b. [12]< G.5–G.6 > Write the VMIPS vector code for the loop without a dependence.

G.6 [20/15/20/20]< G.5–G.6 > The compiled Linpack performance of the Cray-1 (designed in 1976) was almost doubled by a better compiler in 1989. Let's look at a simple example of how this might occur. Consider the DAXPY-like loop (where k is a parameter to the procedure containing the loop):

```
do 10 i=1,64
      do 10 j=1,64
            Y(k,j)=a*X(i,j)+Y(k,j)
10      continue
```

- a. [20]< G.5–G.6 > Write the *straightforward* code sequence for just the inner loop in VMIPS vector instructions.
- b. [15]< G.5–G.6 > Using the techniques of Section G.6, estimate the performance of this code on VMIPS by finding T_{64} in clock cycles. You may assume that T_{loop} of overhead is incurred for each iteration of the outer loop. What limits the performance?
- c. [20]< G.5–G.6 > Rewrite the VMIPS code to reduce the performance limitation; show the resulting inner loop in VMIPS vector instructions. (*Hint:* Think about what establishes T_{chime} ; can you affect it?) Find the total time for the resulting sequence.
- d. [20]< G.5–G.6 > Estimate the performance of your new version, using the techniques of Section G.6 and finding T_{64} .

G.7 [15/15/25]< G.4 > Consider the following code:

```
do 10 i=1,64
      if (B(i).ne.0) then
            A(i)=A(i)/B(i)
10      continue
```

Assume that the addresses of A and B are in Ra and Rb, respectively, and that F0 contains 0.

- a. [15]<G.4> Write the VMIPS code for this loop using the vector-mask capability.
 - b. [15]<G.4> Write the VMIPS code for this loop using scatter-gather.
 - c. [25]<G.4> Estimate the performance (T_{100} in clock cycles) of these two vector loops, assuming a divide latency of 20 cycles. Assume that all vector instructions run at one result per clock, independent of the setting of the vector-mask register. Assume that 50% of the entries of B are 0. Considering hardware costs, which would you build if the above loop were typical?
- G.8 [15/20/15/15]<G.1–G.6> The difference between peak and sustained performance can be large. For one problem, a Hitachi S810 had a peak speed twice as high as that of the Cray X-MP, while for another more realistic problem, the Cray X-MP was twice as fast as the Hitachi processor. Let's examine why this might occur using two versions of VMIPS and the following code sequences:
- ```

C Code sequence 1
do 10 i=1,10000
 A(i)=x * A(i)+y * A(i)
10 continue
C Code sequence 2
do 10 i=1,100
 A(i)=x * A(i)
10 continue

```
- Assume there is a version of VMIPS (call it VMIPS-II) that has two copies of every floating-point functional unit with full chaining among them. Assume that both VMIPS and VMIPS-II have two load-store units. Because of the extra functional units and the increased complexity of assigning operations to units, all the overheads ( $T_{loop}$  and  $T_{start}$ ) are doubled for VMIPS-II.
- a. [15]<G.1–G.6> Find the number of clock cycles on code sequence 1 on VMIPS.
  - b. [20]<G.1–G.6> Find the number of clock cycles on code sequence 1 for VMIPS-II. How does this compare to VMIPS?
  - c. [15]<G.1–G.6> Find the number of clock cycles on code sequence 2 for VMIPS.
  - d. [15]<G.1–G.6> Find the number of clock cycles on code sequence 2 for VMIPS-II. How does this compare to VMIPS?
- G.9 [20]<G.5> Here is a tricky piece of code with two-dimensional arrays. Does this loop have dependences? Can these loops be written so they are parallel? If so, how? Rewrite the *source* code so that it is clear that the loop can be vectorized, if possible.

```

do 290 j=2,n
 do 290 i=2,j
 aa(i,j)=aa(i-1,j)*aa(i-1,j)+bb(i,j)
290 continue

```

G.10 [12/15]< G.5 > Consider the following loop:

```
do 10 i=2,n
 A(i)=B
10 C(i)=A(i - 1)
```

- [12]< G.5 > Show there is a loop-carried dependence in this code fragment.
- [15]< G.5 > Rewrite the code in FORTRAN so that it can be vectorized as two separate vector sequences.

G.11 [15/25/25]< G.5 > As we saw in Section G.5, some loop structures are not easily vectorized. One common structure is a *reduction*—a loop that reduces an array to a single value by repeated application of an operation. This is a special case of a recurrence. A common example occurs in dot product:

```
dot=0.0
do 10 i=1,64
10 dot=dot+A(i) * B(i)
```

This loop has an obvious loop-carried dependence (on *dot*) and cannot be vectorized in a straightforward fashion. The first thing a good vectorizing compiler would do is split the loop to separate out the vectorizable portion and the recurrence and perhaps rewrite the loop as:

```
do 10 i=1,64
10 dot(i)=A(i) * B(i)
 do 20 i=2,64
20 dot(1)=dot(1)+dot(i)
```

The variable *dot* has been expanded into a vector; this transformation is called *scalar expansion*. We can try to vectorize the second loop either relying strictly on the compiler (part (a)) or with hardware support as well (part (b)). There is an important caveat in the use of vector techniques for reduction. To make reduction work, we are relying on the associativity of the operator being used for the reduction. Because of rounding and finite range, however, floating-point arithmetic is not strictly associative. For this reason, most compilers require the programmer to indicate whether associativity can be used to more efficiently compile reductions.

- [15]< G.5 > One simple scheme for compiling the loop with the recurrence is to add sequences of progressively shorter vectors—two 32-element vectors, then two 16-element vectors, and so on. This technique has been called *recursive doubling*. It is faster than doing all the operations in scalar mode. Show how the FORTRAN code would look for execution of the second loop in the preceding code fragment using recursive doubling.
- [25]< G.5 > In some vector processors, the vector registers are addressable, and the operands to a vector operation may be two different parts of the same vector register. This allows another solution for the reduction, called *partial sums*.

The key idea in partial sums is to reduce the vector to  $m$  sums where  $m$  is the total latency through the vector functional unit, including the operand read and write times. Assume that the VMIPS vector registers are addressable (e.g., you can initiate a vector operation with the operand V1(16), indicating that the input operand began with element 16). Also, assume that the total latency for adds, including operand read and write, is eight cycles. Write a VMIPS code sequence that reduces the contents of V1 to eight partial sums. It can be done with one vector operation.

- c. [25]<G.5> Discuss how adding the extension in part (b) would affect a machine that had multiple lanes.
- G.12 [40]<G.3–G.4> Extend the MIPS simulator to be a VMIPS simulator, including the ability to count clock cycles. Write some short benchmark programs in MIPS and VMIPS assembly language. Measure the speedup on VMIPS, the percentage of vectorization, and usage of the functional units.
- G.13 [50]<G.5> Modify the MIPS compiler to include a dependence checker. Run some scientific code and loops through it and measure what percentage of the statements could be vectorized.
- G.14 [Discussion] Some proponents of vector processors might argue that the vector processors have provided the best path to ever-increasing amounts of processor power by focusing their attention on boosting peak vector performance. Others would argue that the emphasis on peak performance is misplaced because an increasing percentage of the programs are dominated by nonvector performance. (Remember Amdahl’s law?) The proponents would respond that programmers should work to make their programs vectorizable. What do you think about this argument?

---

|            |                                                                    |      |
|------------|--------------------------------------------------------------------|------|
| <b>H.1</b> | Introduction: Exploiting Instruction-Level Parallelism Statically  | H-2  |
| <b>H.2</b> | Detecting and Enhancing Loop-Level Parallelism                     | H-2  |
| <b>H.3</b> | Scheduling and Structuring Code for Parallelism                    | H-12 |
| <b>H.4</b> | Hardware Support for Exposing Parallelism: Predicated Instructions | H-23 |
| <b>H.5</b> | Hardware Support for Compiler Speculation                          | H-27 |
| <b>H.6</b> | The Intel IA-64 Architecture and Itanium Processor                 | H-32 |
| <b>H.7</b> | Concluding Remarks                                                 | H-43 |

# H

---

## Hardware and Software for VLIW and EPIC

The EPIC approach is based on the application of massive resources. These resources include more load-store, computational, and branch units, as well as larger, lower-latency caches than would be required for a superscalar processor. Thus, IA-64 gambles that, in the future, power will not be the critical limitation, and that massive resources, along with the machinery to exploit them, will not penalize performance with their adverse effect on clock speed, path length, or CPI factors.

M. Hopkins  
*in a commentary on the EPIC  
approach and the IA-64 architecture (2000)*

## H.1

### Introduction: Exploiting Instruction-Level Parallelism Statically

In this chapter, we discuss compiler technology for increasing the amount of parallelism that we can exploit in a program as well as hardware support for these compiler techniques. The next section defines when a loop is parallel, how a dependence can prevent a loop from being parallel, and techniques for eliminating some types of dependences. The following section discusses the topic of scheduling code to improve parallelism. These two sections serve as an introduction to these techniques.

We do not attempt to explain the details of ILP-oriented compiler techniques, since that would take hundreds of pages, rather than the 20 we have allotted. Instead, we view this material as providing general background that will enable the reader to have a basic understanding of the compiler techniques used to exploit ILP in modern computers.

Hardware support for these compiler techniques can greatly increase their effectiveness, and Sections H.4 and H.5 explore such support. The IA-64 represents the culmination of the compiler and hardware ideas for exploiting parallelism statically and includes support for many of the concepts proposed by researchers during more than a decade of research into the area of compiler-based instruction-level parallelism. Section H.6 provides a description and performance analyses of the Intel IA-64 architecture and its second-generation implementation, Itanium 2.

The core concepts that we exploit in statically based techniques—finding parallelism, reducing control and data dependences, and using speculation—are the same techniques we saw exploited in Chapter 3 using dynamic techniques. The key difference is that the techniques in this appendix are applied at compile time by the compiler, rather than at runtime by the hardware. The advantages of compile time techniques are primarily two: They do not burden runtime execution with any inefficiency, and they can take into account a wider range of the program than a runtime approach might be able to incorporate. As an example of the latter, the next section shows how a compiler might determine that an entire loop can be executed in parallel; hardware techniques might or might not be able to find such parallelism. The major disadvantage of static approaches is that they can use only compile time information. Without runtime information, compile time techniques must often be conservative and assume the worst case.

## H.2

### Detecting and Enhancing Loop-Level Parallelism

Loop-level parallelism is normally analyzed at the source level or close to it, while most analysis of ILP is done once instructions have been generated by the compiler. Loop-level analysis involves determining what dependences exist among the operands in a loop across the iterations of that loop. For now, we will

consider only data dependences, which arise when an operand is written at some point and read at a later point. Name dependences also exist and may be removed by renaming techniques like those we explored in Chapter 3.

The analysis of loop-level parallelism focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations; such a dependence is called a *loop-carried dependence*. Most of the examples we considered in Section 3.2 have no loop-carried dependences and, thus, are loop-level parallel. To see that a loop is parallel, let us first look at the source representation:

```
for (i=1000; i>0; i=i-1)
 x[i] = x[i] + s;
```

In this loop, there is a dependence between the two uses of  $x[i]$ , but this dependence is within a single iteration and is not loop carried. There is a dependence between successive uses of  $i$  in different iterations, which is loop carried, but this dependence involves an induction variable and can be easily recognized and eliminated. We saw examples of how to eliminate dependences involving induction variables during loop unrolling in Section 3.2, and we will look at additional examples later in this section.

Because finding loop-level parallelism involves recognizing structures such as loops, array references, and induction variable computations, the compiler can do this analysis more easily at or near the source level, as opposed to the machine-code level. Let's look at a more complex example.

**Example** Consider a loop like this one:

```
for (i=1; i<=100; i=i+1) {
 A[i+1] = A[i] + C[i]; /* S1 */
 B[i+1] = B[i] + A[i+1]; /* S2 */
}
```

Assume that  $A$ ,  $B$ , and  $C$  are distinct, nonoverlapping arrays. (In practice, the arrays may sometimes be the same or may overlap. Because the arrays may be passed as parameters to a procedure, which includes this loop, determining whether arrays overlap or are identical often requires sophisticated, interprocedural analysis of the program.) What are the data dependences among the statements S1 and S2 in the loop?

**Answer** There are two different dependences:

1. S1 uses a value computed by S1 in an earlier iteration, since iteration  $i$  computes  $A[i+1]$ , which is read in iteration  $i+1$ . The same is true of S2 for  $B[i]$  and  $B[i+1]$ .
2. S2 uses the value,  $A[i+1]$ , computed by S1 in the same iteration.

These two dependences are different and have different effects. To see how they differ, let's assume that only one of these dependences exists at a time. Because the dependence of statement S1 is on an earlier iteration of S1, this dependence is loop carried. This dependence forces successive iterations of this loop to execute in series.

The second dependence (S2 depending on S1) is within an iteration and is not loop carried. Thus, if this were the only dependence, multiple iterations of the loop could execute in parallel, as long as each pair of statements in an iteration were kept in order. We saw this type of dependence in an example in Section 3.2, where unrolling was able to expose the parallelism.

It is also possible to have a loop-carried dependence that does not prevent parallelism, as the next example shows.

---

---

**Example** Consider a loop like this one:

```
for (i=1; i<=100; i=i+1) {
 A[i] = A[i] + B[i]; /* S1 */
 B[i+1] = C[i] + D[i]; /* S2 */
}
```

What are the dependences between S1 and S2? Is this loop parallel? If not, show how to make it parallel.

**Answer** Statement S1 uses the value assigned in the previous iteration by statement S2, so there is a loop-carried dependence between S2 and S1. Despite this loop-carried dependence, this loop can be made parallel. Unlike the earlier loop, this dependence is not circular: Neither statement depends on itself, and, although S1 depends on S2, S2 does not depend on S1. A loop is parallel if it can be written without a cycle in the dependences, since the absence of a cycle means that the dependences give a partial ordering on the statements.

Although there are no circular dependences in the above loop, it must be transformed to conform to the partial ordering and expose the parallelism. Two observations are critical to this transformation:

1. There is no dependence from S1 to S2. If there were, then there would be a cycle in the dependences and the loop would not be parallel. Since this other dependence is absent, interchanging the two statements will not affect the execution of S2.
2. On the first iteration of the loop, statement S1 depends on the value of B[1] computed prior to initiating the loop.

These two observations allow us to replace the loop above with the following code sequence:

```

A[1] = A[1] + B[1];
for (i=1; i<=99; i=i+1) {
 B[i+1] = C[i] + D[i];
 A[i+1] = A[i+1] + B[i+1];
}
B[101] = C[100] + D[100];

```

The dependence between the two statements is no longer loop carried, so iterations of the loop may be overlapped, provided the statements in each iteration are kept in order.

---

Our analysis needs to begin by finding all loop-carried dependences. This dependence information is *inexact*, in the sense that it tells us that such a dependence *may* exist. Consider the following example:

```

for (i=1; i<=100; i=i+1) {
 A[i] = B[i] + C[i]
 D[i] = A[i] * E[i]
}

```

The second reference to A in this example need not be translated to a load instruction, since we know that the value is computed and stored by the previous statement; hence, the second reference to A can simply be a reference to the register into which A was computed. Performing this optimization requires knowing that the two references are *always* to the same memory address and that there is no intervening access to the same location. Normally, data dependence analysis only tells that one reference *may* depend on another; a more complex analysis is required to determine that two references *must be* to the exact same address. In the example above, a simple version of this analysis suffices, since the two references are in the same basic block.

Often loop-carried dependences are in the form of a *recurrence*:

```

for (i=2; i<=100; i=i+1) {
 Y[i] = Y[i-1] + Y[i];
}

```

A recurrence is when a variable is defined based on the value of that variable in an earlier iteration, often the one immediately preceding, as in the above fragment. Detecting a recurrence can be important for two reasons: Some architectures (especially vector computers) have special support for executing recurrences, and some recurrences can be the source of a reasonable amount of parallelism. To see how the latter can be true, consider this loop:

```

for (i=6; i<=100; i=i+1) {
 Y[i] = Y[i-5] + Y[i];
}

```

On the iteration  $i$ , the loop references element  $i - 5$ . The loop is said to have a *dependence distance* of 5. Many loops with carried dependences have a dependence distance of 1. The larger the distance, the more potential parallelism can be obtained by unrolling the loop. For example, if we unroll the first loop, with a dependence distance of 1, successive statements are dependent on one another; there is still some parallelism among the individual instructions, but not much. If we unroll the loop that has a dependence distance of 5, there is a sequence of five statements that have no dependences, and thus much more ILP. Although many loops with loop-carried dependences have a dependence distance of 1, cases with larger distances do arise, and the longer distance may well provide enough parallelism to keep a processor busy.

## Finding Dependences

Finding the dependences in a program is an important part of three tasks: (1) good scheduling of code, (2) determining which loops might contain parallelism, and (3) eliminating name dependences. The complexity of dependence analysis arises because of the presence of arrays and pointers in languages like C or C++, or pass-by-reference parameter passing in FORTRAN. Since scalar variable references explicitly refer to a name, they can usually be analyzed quite easily, with aliasing because of pointers and reference parameters causing some complications and uncertainty in the analysis.

How does the compiler detect dependences in general? Nearly all dependence analysis algorithms work on the assumption that array indices are *affine*. In simplest terms, a one-dimensional array index is affine if it can be written in the form  $a \times i + b$ , where  $a$  and  $b$  are constants and  $i$  is the loop index variable. The index of a multidimensional array is affine if the index in each dimension is affine. Sparse array accesses, which typically have the form  $\times[y[i]]$ , are one of the major examples of nonaffine accesses.

Determining whether there is a dependence between two references to the same array in a loop is thus equivalent to determining whether two affine functions can have the same value for different indices between the bounds of the loop. For example, suppose we have stored to an array element with index value  $a \times i + b$  and loaded from the same array with index value  $c \times i + d$ , where  $i$  is the for-loop index variable that runs from  $m$  to  $n$ . A dependence exists if two conditions hold:

1. There are two iteration indices,  $j$  and  $k$ , both within the limits of the for loop. That is,  $m \leq j \leq n$ ,  $m \leq k \leq n$ .
2. The loop stores into an array element indexed by  $a \times j + b$  and later fetches from that *same* array element when it is indexed by  $c \times k + d$ . That is,  $a \times j + b = c \times k + d$ .

In general, we cannot determine whether a dependence exists at compile time. For example, the values of  $a$ ,  $b$ ,  $c$ , and  $d$  may not be known (they could be values in other arrays), making it impossible to tell if a dependence exists. In other cases, the dependence testing may be very expensive but decidable at compile time. For example, the accesses may depend on the iteration indices of multiple nested loops. Many programs, however, contain primarily simple indices where  $a$ ,  $b$ ,  $c$ , and  $d$  are all constants. For these cases, it is possible to devise reasonable compile time tests for dependence.

As an example, a simple and sufficient test for the absence of a dependence is the *greatest common divisor* (GCD) test. It is based on the observation that if a loop-carried dependence exists, then  $\text{GCD}(c,a)$  must divide  $(d - b)$ . (Recall that an integer,  $x$ , *divides* another integer,  $y$ , if we get an integer quotient when we do the division  $y/x$  and there is no remainder.)

**Example** Use the GCD test to determine whether dependences exist in the following loop:

```
for (i=1; i<=100; i=i+1) {
 X[2*i+3] = X[2*i] * 5.0;
}
```

**Answer** Given the values  $a = 2$ ,  $b = 3$ ,  $c = 2$ , and  $d = 0$ , then  $\text{GCD}(a,c) = 2$ , and  $d - b = -3$ . Since 2 does not divide  $-3$ , no dependence is possible.

---

The GCD test is sufficient to guarantee that no dependence exists; however, there are cases where the GCD test succeeds but no dependence exists. This can arise, for example, because the GCD test does not take the loop bounds into account.

In general, determining whether a dependence actually exists is NP complete. In practice, however, many common cases can be analyzed precisely at low cost. Recently, approaches using a hierarchy of exact tests increasing in generality and cost have been shown to be both accurate and efficient. (A test is *exact* if it precisely determines whether a dependence exists. Although the general case is NP complete, there exist exact tests for restricted situations that are much cheaper.)

In addition to detecting the presence of a dependence, a compiler wants to classify the type of dependence. This classification allows a compiler to recognize name dependences and eliminate them at compile time by renaming and copying.

**Example** The following loop has multiple types of dependences. Find all the true dependences, output dependences, and antidependences, and eliminate the output dependences and antidependences by renaming.

```
for (i=1; i<=100; i=i+1) {
 Y[i] = X[i] / c; /* S1 */
 X[i] = X[i] + c; /* S2 */
 Z[i] = Y[i] + c; /* S3 */
 Y[i] = c - Y[i]; /* S4 */
}
```

**Answer** The following dependences exist among the four statements:

1. There are true dependences from S1 to S3 and from S1 to S4 because of  $Y[i]$ . These are not loop carried, so they do not prevent the loop from being considered parallel. These dependences will force S3 and S4 to wait for S1 to complete.
2. There is an antidependence from S1 to S2, based on  $X[i]$ .
3. There is an antidependence from S3 to S4 for  $Y[i]$ .
4. There is an output dependence from S1 to S4, based on  $Y[i]$ .

The following version of the loop eliminates these false (or pseudo) dependences:

```
for (i=1; i<=100; i=i+1 {
 /* Y renamed to T to remove output dependence */
 T[i] = X[i] / c;
 /* X renamed to X1 to remove antidependence */
 X1[i] = X[i] + c;
 /* Y renamed to T to remove antidependence */
 Z[i] = T[i] + c;
 Y[i] = c - T[i];
}
```

After the loop, the variable  $X$  has been renamed  $X1$ . In code that follows the loop, the compiler can simply replace the name  $X$  by  $X1$ . In this case, renaming does not require an actual copy operation but can be done by substituting names or by register allocation. In other cases, however, renaming will require copying.

---

Dependence analysis is a critical technology for exploiting parallelism. At the instruction level, it provides information needed to interchange memory references when scheduling, as well as to determine the benefits of unrolling a loop. For detecting loop-level parallelism, dependence analysis is the basic tool. Effectively compiling programs to either vector computers or multiprocessors depends critically on this analysis. The major drawback of dependence analysis is that it applies only under a limited set of circumstances—namely, among references within a single loop nest and using affine index functions. Thus, there is a wide variety of situations in which array-oriented dependence analysis *cannot* tell us what we might want to know, including the following:

- When objects are referenced via pointers rather than array indices (but see discussion below)
- When array indexing is indirect through another array, which happens with many representations of sparse arrays
- When a dependence may exist for some value of the inputs but does not exist in actuality when the code is run since the inputs never take on those values
- When an optimization depends on knowing more than just the possibility of a dependence but needs to know on *which* write of a variable does a read of that variable depend

To deal with the issue of analyzing programs with pointers, another type of analysis, often called *points-to* analysis, is required (see Wilson and Lam [1995]). The key question that we want answered from dependence analysis of pointers is whether two pointers can designate the same address. In the case of complex dynamic data structures, this problem is extremely difficult. For example, we may want to know whether two pointers can reference the *same* node in a list at a given point in a program, which in general is undecidable and in practice is extremely difficult to answer. We may, however, be able to answer a simpler question: Can two pointers designate nodes in the *same* list, even if they may be separate nodes? This more restricted analysis can still be quite useful in scheduling memory accesses performed through pointers.

The basic approach used in points-to analysis relies on information from three major sources:

1. Type information, which restricts what a pointer can point to.
2. Information derived when an object is allocated or when the address of an object is taken, which can be used to restrict what a pointer can point to. For example, if  $p$  always points to an object allocated in a given source line and  $q$  never points to that object, then  $p$  and  $q$  can never point to the same object.
3. Information derived from pointer assignments. For example, if  $p$  may be assigned the value of  $q$ , then  $p$  may point to anything  $q$  points to.

There are several cases where analyzing pointers has been successfully applied and is extremely useful:

- When pointers are used to pass the address of an object as a parameter, it is possible to use points-to analysis to determine the possible set of objects referenced by a pointer. One important use is to determine if two pointer parameters may designate the same object.
- When a pointer can point to one of several types, it is sometimes possible to determine the type of the data object that a pointer designates at different parts of the program.
- It is often possible to separate out pointers that may only point to a local object versus a global one.

There are two different types of limitations that affect our ability to do accurate dependence analysis for large programs. The first type of limitation arises from restrictions in the analysis algorithms. Often, we are limited by the lack of applicability of the analysis rather than a shortcoming in dependence analysis *per se*. For example, dependence analysis for pointers is essentially impossible for programs that use pointers in arbitrary fashion—such as by doing arithmetic on pointers.

The second limitation is the need to analyze behavior across procedure boundaries to get accurate information. For example, if a procedure accepts two parameters that are pointers, determining whether the values could be the same requires analyzing across procedure boundaries. This type of analysis, called *interprocedural analysis*, is much more difficult and complex than analysis within a single procedure. Unlike the case of analyzing array indices within a single loop nest, points-to analysis usually requires an interprocedural analysis. The reason for this is simple. Suppose we are analyzing a program segment with two pointers; if the analysis does not know anything about the two pointers at the start of the program segment, it must be conservative and assume the worst case. The worst case is that the two pointers *may* designate the same object, but they are not *guaranteed* to designate the same object. This worst case is likely to propagate through the analysis, producing useless information. In practice, getting fully accurate interprocedural information is usually too expensive for real programs. Instead, compilers usually use approximations in interprocedural analysis. The result is that the information may be too inaccurate to be useful.

Modern programming languages that use strong typing, such as Java, make the analysis of dependences easier. At the same time the extensive use of procedures to structure programs, as well as abstract data types, makes the analysis more difficult. Nonetheless, we expect that continued advances in analysis algorithms, combined with the increasing importance of pointer dependency analysis, will mean that there is continued progress on this important problem.

## Eliminating Dependent Computations

Compilers can reduce the impact of dependent computations so as to achieve more instruction-level parallelism (ILP). The key technique is to eliminate or reduce a dependent computation by back substitution, which increases the amount of parallelism and sometimes increases the amount of computation required. These techniques can be applied both within a basic block and within loops, and we describe them differently.

Within a basic block, algebraic simplifications of expressions and an optimization called *copy propagation*, which eliminates operations that copy values, can be used to simplify sequences like the following:

```
DADDUI R1,R2,#4
DADDUI R1,R1,#4
```

to

DADDUI R1,R2,#8

assuming this is the only use of R1. In fact, the techniques we used to reduce multiple increments of array indices during loop unrolling and to move the increments across memory addresses in Section 3.2 are examples of this type of optimization.

In these examples, computations are actually eliminated, but it is also possible that we may want to increase the parallelism of the code, possibly even increasing the number of operations. Such optimizations are called *tree height reduction* because they reduce the height of the tree structure representing a computation, making it wider but shorter. Consider the following code sequence:

|     |          |
|-----|----------|
| ADD | R1,R2,R3 |
| ADD | R4,R1,R6 |
| ADD | R8,R4,R7 |

Notice that this sequence requires at least three execution cycles, since all the instructions depend on the immediate predecessor. By taking advantage of associativity, we can transform the code and rewrite it as

|     |          |
|-----|----------|
| ADD | R1,R2,R3 |
| ADD | R4,R6,R7 |
| ADD | R8,R1,R4 |

This sequence can be computed in two execution cycles. When loop unrolling is used, opportunities for these types of optimizations occur frequently.

Although arithmetic with unlimited range and precision is associative, computer arithmetic is not associative, for either integer arithmetic, because of limited range, or floating-point arithmetic, because of both range and precision. Thus, using these restructuring techniques can sometimes lead to erroneous behavior, although such occurrences are rare. For this reason, most compilers require that optimizations that rely on associativity be explicitly enabled.

When loops are unrolled, this sort of algebraic optimization is important to reduce the impact of dependences arising from recurrences. *Recurrences* are expressions whose value on one iteration is given by a function that depends on the previous iterations. When a loop with a recurrence is unrolled, we may be able to algebraically optimize the unrolled loop, so that the recurrence need only be evaluated once per unrolled iteration. One common type of recurrence arises from an explicit program statement, such as:

sum = sum + x;

Assume we unroll a loop with this recurrence five times. If we let the value of  $x$  on these five iterations be given by  $x_1, x_2, x_3, x_4$ , and  $x_5$ , then we can write the value of  $\text{sum}$  at the end of each unroll as:

$$\text{sum} = \text{sum} + x_1 + x_2 + x_3 + x_4 + x_5;$$

If unoptimized, this expression requires five dependent operations, but it can be rewritten as:

$$\text{sum} = ((\text{sum} + x_1) + (x_2 + x_3)) + (x_4 + x_5);$$

which can be evaluated in only three dependent operations.

Recurrences also arise from implicit calculations, such as those associated with array indexing. Each array index translates to an address that is computed based on the loop index variable. Again, with unrolling and algebraic optimization, the dependent computations can be minimized.

### H.3

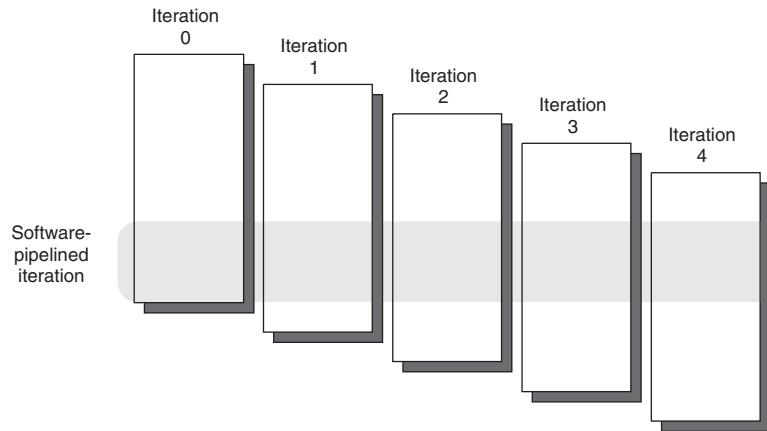
## Scheduling and Structuring Code for Parallelism

We have already seen that one compiler technique, loop unrolling, is useful to uncover parallelism among instructions by creating longer sequences of straight-line code. There are two other important techniques that have been developed for this purpose: *software pipelining* and *trace scheduling*.

### Software Pipelining: Symbolic Loop Unrolling

*Software pipelining* is a technique for reorganizing loops such that each iteration in the software-pipelined code is made from instructions chosen from different iterations of the original loop. This approach is most easily understood by looking at the scheduled code for the unrolled loop, which appeared in the example in Section 2.2. The scheduler in this example essentially interleaves instructions from different loop iterations, so as to separate the dependent instructions that occur within a single loop iteration. By choosing instructions from different iterations, dependent computations are separated from one another by an entire loop body, increasing the possibility that the unrolled loop can be scheduled without stalls.

A software-pipelined loop interleaves instructions from different iterations without unrolling the loop, as illustrated in Figure H.1. This technique is the software counterpart to what Tomasulo's algorithm does in hardware. The software-pipelined loop for the earlier example would contain one load, one add, and one store, each from a different iteration. There is also some start-up code that is needed before the loop begins as well as code to finish up after the loop is completed. We will ignore these in this discussion, for simplicity.



**Figure H.1** A software-pipelined loop chooses instructions from different loop iterations, thus separating the dependent instructions within one iteration of the original loop. The start-up and finish-up code will correspond to the portions above and below the software-pipelined iteration.

---

**Example** Show a software-pipelined version of this loop, which increments all the elements of an array whose starting address is in R1 by the contents of F2:

```
Loop: L.D F0,0(R1)
 ADD.D F4,F0,F2
 S.D F4,0(R1)
 DADDUI R1,R1,#-8
 BNE R1,R2,Loop
```

You may omit the start-up and clean-up code.

**Answer** Software pipelining symbolically unrolls the loop and then selects instructions from each iteration. Since the unrolling is symbolic, the loop overhead instructions (the DADDUI and BNE) need not be replicated. Here's the body of the unrolled loop without overhead instructions, highlighting the instructions taken from each iteration:

|                |       |          |
|----------------|-------|----------|
| Iteration i:   | L.D   | F0,0(R1) |
|                | ADD.D | F4,F0,F2 |
|                | S.D   | F4,0(R1) |
| Iteration i+1: | L.D   | F0,0(R1) |
|                | ADD.D | F4,F0,F2 |
|                | S.D   | F4,0(R1) |
| Iteration i+2: | L.D   | F0,0(R1) |
|                | ADD.D | F4,F0,F2 |
|                | S.D   | F4,0(R1) |

The selected instructions from different iterations are then put together in the loop with the loop control instructions:

|       |        |            |                    |
|-------|--------|------------|--------------------|
| Loop: | S.D    | F4,16(R1)  | ; stores into M[i] |
|       | ADD.D  | F4,F0,F2   | ; adds to M[i-1]   |
|       | L.D    | F0,0(R1)   | ; loads M[i-2]     |
|       | DADDUI | R1,R1,#-8  |                    |
|       | BNE    | R1,R2,Loop |                    |

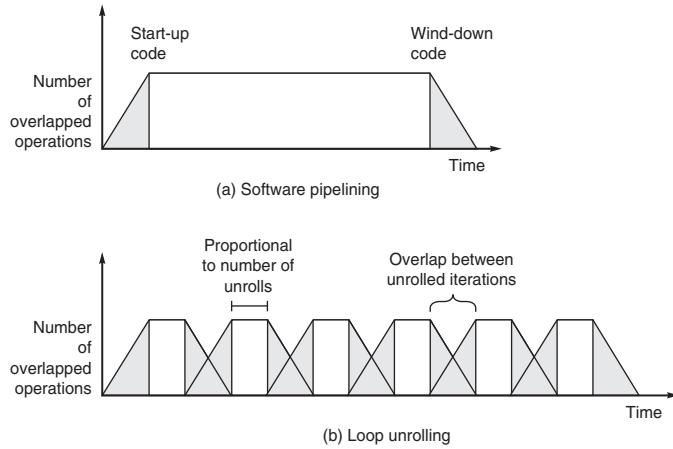
This loop can be run at a rate of 5 cycles per result, ignoring the start-up and clean-up portions, and assuming that DADDUI is scheduled before the ADD.D and that the L.D instruction, with an adjusted offset, is placed in the branch delay slot. Because the load and store are separated by offsets of 16 (two iterations), the loop should run for two fewer iterations. Notice that the reuse of registers (e.g., F4, F0, and R1) requires the hardware to avoid the write after read (WAR) hazards that would occur in the loop. This hazard should not be a problem in this case, since no data-dependent stalls should occur.

By looking at the unrolled version we can see what the start-up code and finish-up code will need to be. For start-up, we will need to execute any instructions that correspond to iteration 1 and 2 that will not be executed. These instructions are the L.D for iterations 1 and 2 and the ADD.D for iteration 1. For the finish-up code, we need to execute any instructions that will not be executed in the final two iterations. These include the ADD.D for the last iteration and the S.D for the last two iterations.

---

Register management in software-pipelined loops can be tricky. The previous example is not too hard since the registers that are written on one loop iteration are read on the next. In other cases, we may need to increase the number of iterations between when we issue an instruction and when the result is used. This increase is required when there are a small number of instructions in the loop body and the latencies are large. In such cases, a combination of software pipelining and loop unrolling is needed.

Software pipelining can be thought of as *symbolic* loop unrolling. Indeed, some of the algorithms for software pipelining use loop-unrolling algorithms to figure out how to software-pipeline the loop. The major advantage of software pipelining over straight loop unrolling is that software pipelining consumes less code space. Software pipelining and loop unrolling, in addition to yielding a better scheduled inner loop, each reduce a different type of overhead. Loop unrolling reduces the overhead of the loop—the branch and counter update code. Software pipelining reduces the time when the loop is not running at peak speed to once per loop at the beginning and end. If we unroll a loop that does 100 iterations a constant number of times, say, 4, we pay the overhead  $100/4 = 25$  times—every time the inner unrolled loop is initiated. Figure H.2 shows this behavior graphically. Because these techniques attack two different types of overhead, the best performance can come from doing both. In practice, compilation using software pipelining is quite difficult for several reasons: Many loops require significant transformation



**Figure H.2** The execution pattern for (a) a software-pipelined loop and (b) an unrolled loop. The shaded areas are the times when the loop is not running with maximum overlap or parallelism among instructions. This occurs once at the beginning and once at the end for the software-pipelined loop. For the unrolled loop it occurs  $m/n$  times if the loop has a total of  $m$  iterations and is unrolled  $n$  times. Each block represents an unroll of  $n$  iterations. Increasing the number of unrollings will reduce the start-up and clean-up overhead. The overhead of one iteration overlaps with the overhead of the next, thereby reducing the impact. The total area under the polygonal region in each case will be the same, since the total number of operations is just the execution rate multiplied by the time.

before they can be software pipelined, the trade-offs in terms of overhead versus efficiency of the software-pipelined loop are complex, and the issue of register management creates additional complexities. To help deal with the last two of these issues, the IA-64 added extensive hardware support for software pipelining. Although this hardware can make it more efficient to apply software pipelining, it does not eliminate the need for complex compiler support, or the need to make difficult decisions about the best way to compile a loop.

### Global Code Scheduling

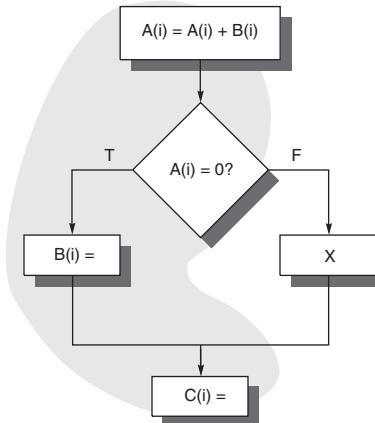
In Section 3.2 we examined the use of loop unrolling and code scheduling to improve ILP. The techniques in Section 3.2 work well when the loop body is straight-line code, since the resulting unrolled loop looks like a single basic block. Similarly, software pipelining works well when the body is a single basic block, since it is easier to find the repeatable schedule. When the body of an unrolled loop contains internal control flow, however, scheduling the code is much more complex. In general, effective scheduling of a loop body with internal control flow will require moving instructions across branches, which is global code scheduling. In this section, we first examine the challenge and limitations of global code

scheduling. In Section H.4 we examine hardware support for eliminating control flow within an inner loop, then we examine two compiler techniques that can be used when eliminating the control flow is not a viable approach.

Global code scheduling aims to compact a code fragment with internal control structure into the shortest possible sequence that preserves the data and control dependences. The data dependences force a partial order on operations, while the control dependences dictate instructions across which code cannot be easily moved. Data dependences are overcome by unrolling and, in the case of memory operations, using dependence analysis to determine if two references refer to the same address. Finding the shortest possible sequence for a piece of code means finding the shortest sequence for the *critical path*, which is the longest sequence of dependent instructions.

Control dependences arising from loop branches are reduced by unrolling. Global code scheduling can reduce the effect of control dependences arising from conditional nonloop branches by moving code. Since moving code across branches will often affect the frequency of execution of such code, effectively using global code motion requires estimates of the relative frequency of different paths. Although global code motion cannot guarantee faster code, if the frequency information is accurate, the compiler can determine whether such code movement is likely to lead to faster code.

Global code motion is important since many inner loops contain conditional statements. Figure H.3 shows a typical code fragment, which may be thought of as an iteration of an unrolled loop, and highlights the more common control flow.




---

**Figure H.3** A code fragment and the common path shaded with gray. Moving the assignments to B or C requires a more complex analysis than for straight-line code. In this section we focus on scheduling this code segment efficiently without hardware assistance. Predication or conditional instructions, which we discuss in the next section, provide another way to schedule this code.

Effectively scheduling this code could require that we move the assignments to B and C to earlier in the execution sequence, before the test of A. Such global code motion must satisfy a set of constraints to be legal. In addition, the movement of the code associated with B, unlike that associated with C, is speculative: It will speed the computation up only when the path containing the code would be taken.

To perform the movement of B, we must ensure that neither the data flow nor the exception behavior is changed. Compilers avoid changing the exception behavior by not moving certain classes of instructions, such as memory references, that can cause exceptions. In Section H.5, we will see how hardware support allows for more opportunities for speculative code motion and removes control dependences. Although such enhanced support for speculation can make it possible to explore more opportunities, the difficulty of choosing how to best compile the code remains complex.

How can the compiler ensure that the assignments to B and C can be moved without affecting the data flow? To see what's involved, let's look at a typical code generation sequence for the flowchart in Figure H.3. Assuming that the addresses for A, B, C are in R1, R2, and R3, respectively, here is such a sequence:

```

LD R4,0(R1) ;load A
LD R5,0(R2) ;load B
DADDU R4,R4,R5 ;Add to A
SD R4,0(R1) ;Store A
...
BNEZ R4,elsepart ;Test A
...
SD ...,0(R2) ;Stores to B
...
J join ;jump over else
elsepart:
...
X join ;else part
 ;code for X
...
join:
...
SD ...,0(R3) ;after if
 ;store C[i]

```

Let's first consider the problem of moving the assignment to B to before the BNEZ instruction. Call the last instruction to assign to B before the if statement *i*. If B is referenced before it is assigned either in code segment X or after the if statement, call the referencing instruction *j*. If there is such an instruction *j*, then moving the assignment to B will change the data flow of the program. In particular, moving the assignment to B will cause *j* to become data dependent on the moved version of the assignment to B rather than on *i*, on which *j* originally depended. You could imagine more clever schemes to allow B to be moved even when the value is used: For example, in the first case, we could make a shadow copy of B before the if statement and use that shadow copy in X. Such schemes are usually avoided, both because they are complex to implement and because they will

slow down the program if the trace selected is not optimal and the operations end up requiring additional instructions to execute.

Moving the assignment to C up to before the first branch requires two steps. First, the assignment is moved over the join point of the else part into the portion corresponding to the then part. This movement makes the instructions for C control dependent on the branch and means that they will not execute if the else path, which is the infrequent path, is chosen. Hence, instructions that were data dependent on the assignment to C, and which execute after this code fragment, will be affected. To ensure the correct value is computed for such instructions, a copy is made of the instructions that compute and assign to C on the else path. Second, we can move C from the then part of the branch across the branch condition, if it does not affect any data flow into the branch condition. If C is moved to before the if test, the copy of C in the else branch can usually be eliminated, since it will be redundant.

We can see from this example that global code scheduling is subject to many constraints. This observation is what led designers to provide hardware support to make such code motion easier, and Sections H.4 and H.5 explores such support in detail.

Global code scheduling also requires complex trade-offs to make code motion decisions. For example, assuming that the assignment to B can be moved before the conditional branch (possibly with some compensation code on the alternative branch), will this movement make the code run faster? The answer is, possibly! Similarly, moving the copies of C into the if and else branches makes the code initially bigger! Only if the compiler can successfully move the computation across the if test will there be a likely benefit.

Consider the factors that the compiler would have to consider in moving the computation and assignment of B:

- What are the relative execution frequencies of the then case and the else case in the branch? If the then case is much more frequent, the code motion may be beneficial. If not, it is less likely, although not impossible, to consider moving the code.
- What is the cost of executing the computation and assignment to B above the branch? It may be that there are a number of empty instruction issue slots in the code above the branch and that the instructions for B can be placed into these slots that would otherwise go empty. This opportunity makes the computation of B “free” at least to first order.
- How will the movement of B change the execution time for the then case? If B is at the start of the critical path for the then case, moving it may be highly beneficial.
- Is B the best code fragment that can be moved above the branch? How does it compare with moving C or other statements within the then case?
- What is the cost of the compensation code that may be necessary for the else case? How effectively can this code be scheduled, and what is its impact on execution time?

As we can see from this *partial* list, global code scheduling is an extremely complex problem. The trade-offs depend on many factors, and individual decisions to globally schedule instructions are highly interdependent. Even choosing which instructions to start considering as candidates for global code motion is complex!

To try to simplify this process, several different methods for global code scheduling have been developed. The two methods we briefly explore here rely on a simple principle: Focus the attention of the compiler on a straight-line code segment representing what is estimated to be the most frequently executed code path. Unrolling is used to generate the straight-line code, but, of course, the complexity arises in how conditional branches are handled. In both cases, they are effectively straightened by choosing and scheduling the most frequent path.

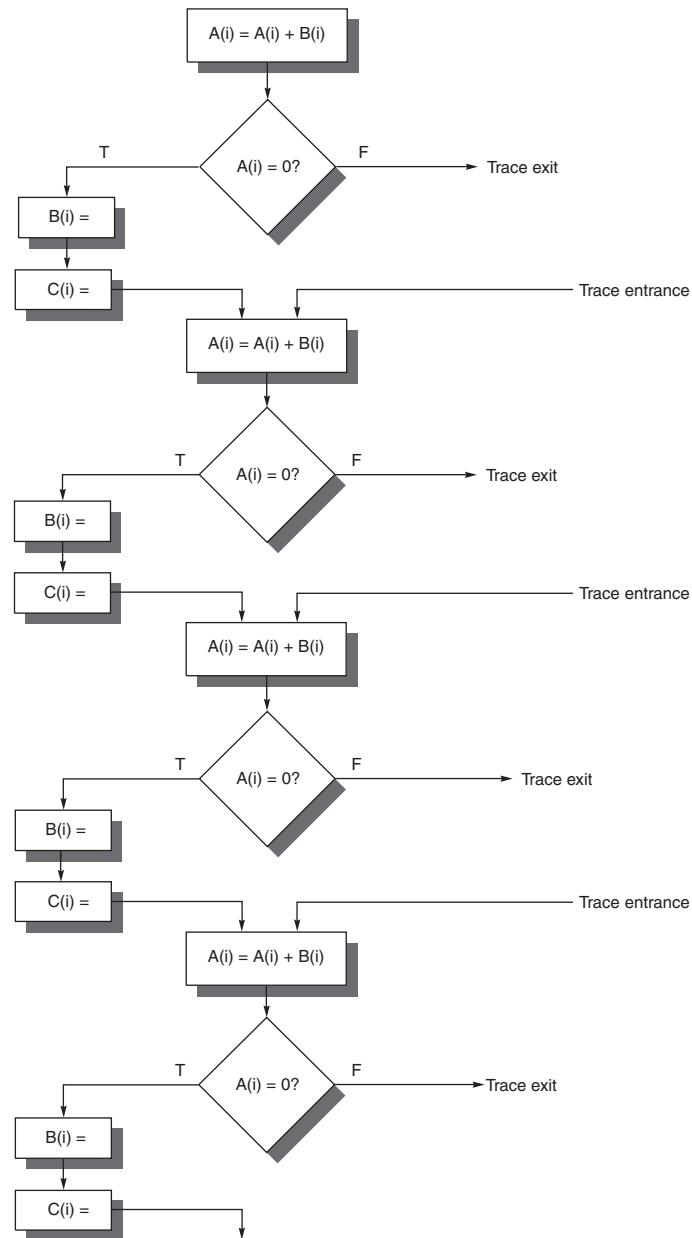
#### *Trace Scheduling: Focusing on the Critical Path*

Trace scheduling is useful for processors with a large number of issues per clock, where conditional or predicated execution (see Section H.4) is inappropriate or unsupported, and where simple loop unrolling may not be sufficient by itself to uncover enough ILP to keep the processor busy. Trace scheduling is a way to organize the global code motion process, so as to simplify the code scheduling by incurring the costs of possible code motion on the less frequent paths. Because it can generate *significant* overheads on the designated infrequent path, it is best used where profile information indicates significant differences in frequency between different paths and where the profile information is highly indicative of program behavior independent of the input. Of course, this limits its effective applicability to certain classes of programs.

There are two steps to trace scheduling. The first step, called *trace selection*, tries to find a likely sequence of basic blocks whose operations will be put together into a smaller number of instructions; this sequence is called a *trace*. Loop unrolling is used to generate long traces, since loop branches are taken with high probability. Additionally, by using static branch prediction, other conditional branches are also chosen as taken or not taken, so that the resultant trace is a straight-line sequence resulting from concatenating many basic blocks. If, for example, the program fragment shown in Figure H.3 corresponds to an inner loop with the highlighted path being much more frequent, and the loop were unwound four times, the primary trace would consist of four copies of the shaded portion of the program, as shown in Figure H.4.

Once a trace is selected, the second process, called *trace compaction*, tries to squeeze the trace into a small number of wide instructions. Trace compaction is code scheduling; hence, it attempts to move operations as early as it can in a sequence (trace), packing the operations into as few wide instructions (or issue packets) as possible.

The advantage of the trace scheduling approach is that it simplifies the decisions concerning global code motion. In particular, branches are viewed as jumps into or out of the selected trace, which is assumed to be the most probable path.



**Figure H.4** This trace is obtained by assuming that the program fragment in Figure H.3 is the inner loop and unwinding it four times, treating the shaded portion in Figure H.3 as the likely path. The trace exits correspond to jumps off the frequent path, and the trace entrances correspond to returns to the trace.

When code is moved across such trace entry and exit points, additional bookkeeping code will often be needed on the entry or exit point. The key assumption is that the trace is so much more probable than the alternatives that the cost of the bookkeeping code need not be a deciding factor: If an instruction can be moved and thereby make the main trace execute faster, it is moved.

Although trace scheduling has been successfully applied to scientific code with its intensive loops and accurate profile data, it remains unclear whether this approach is suitable for programs that are less simply characterized and less loop intensive. In such programs, the significant overheads of compensation code may make trace scheduling an unattractive approach, or, at best, its effective use will be extremely complex for the compiler.

### *Superblocks*

One of the major drawbacks of trace scheduling is that the entries and exits into the middle of the trace cause significant complications, requiring the compiler to generate and track the compensation code and often making it difficult to assess the cost of such code. *Superblocks* are formed by a process similar to that used for traces, but are a form of extended basic blocks, which are restricted to a single entry point but allow multiple exits.

Because superblocks have only a single entry point, compacting a superblock is easier than compacting a trace since only code motion across an exit need be considered. In our earlier example, we would form superblocks that contained only one entrance; hence, moving C would be easier. Furthermore, in loops that have a single loop exit based on a count (for example, a for loop with no loop exit other than the loop termination condition), the resulting superblocks have only one exit as well as one entrance. Such blocks can then be scheduled more easily.

How can a superblock with only one entrance be constructed? The answer is to use *tail duplication* to create a separate block that corresponds to the portion of the trace after the entry. In our previous example, each unrolling of the loop would create an exit from the superblock to a residual loop that handles the remaining iterations. Figure H.5 shows the superblock structure if the code fragment from Figure H.3 is treated as the body of an inner loop and unrolled four times. The residual loop handles any iterations that occur if the superblock is exited, which, in turn, occurs when the unpredicted path is selected. If the expected frequency of the residual loop were still high, a superblock could be created for that loop as well.

The superblock approach reduces the complexity of bookkeeping and scheduling versus the more general trace generation approach but may enlarge code size more than a trace-based approach. Like trace scheduling, superblock scheduling may be most appropriate when other techniques (e.g., if conversion) fail. Even in such cases, assessing the cost of code duplication may limit the usefulness of the approach and will certainly complicate the compilation process.

Loop unrolling, software pipelining, trace scheduling, and superblock scheduling all aim at trying to increase the amount of ILP that can be exploited by a processor issuing more than one instruction on every clock cycle. The effectiveness of

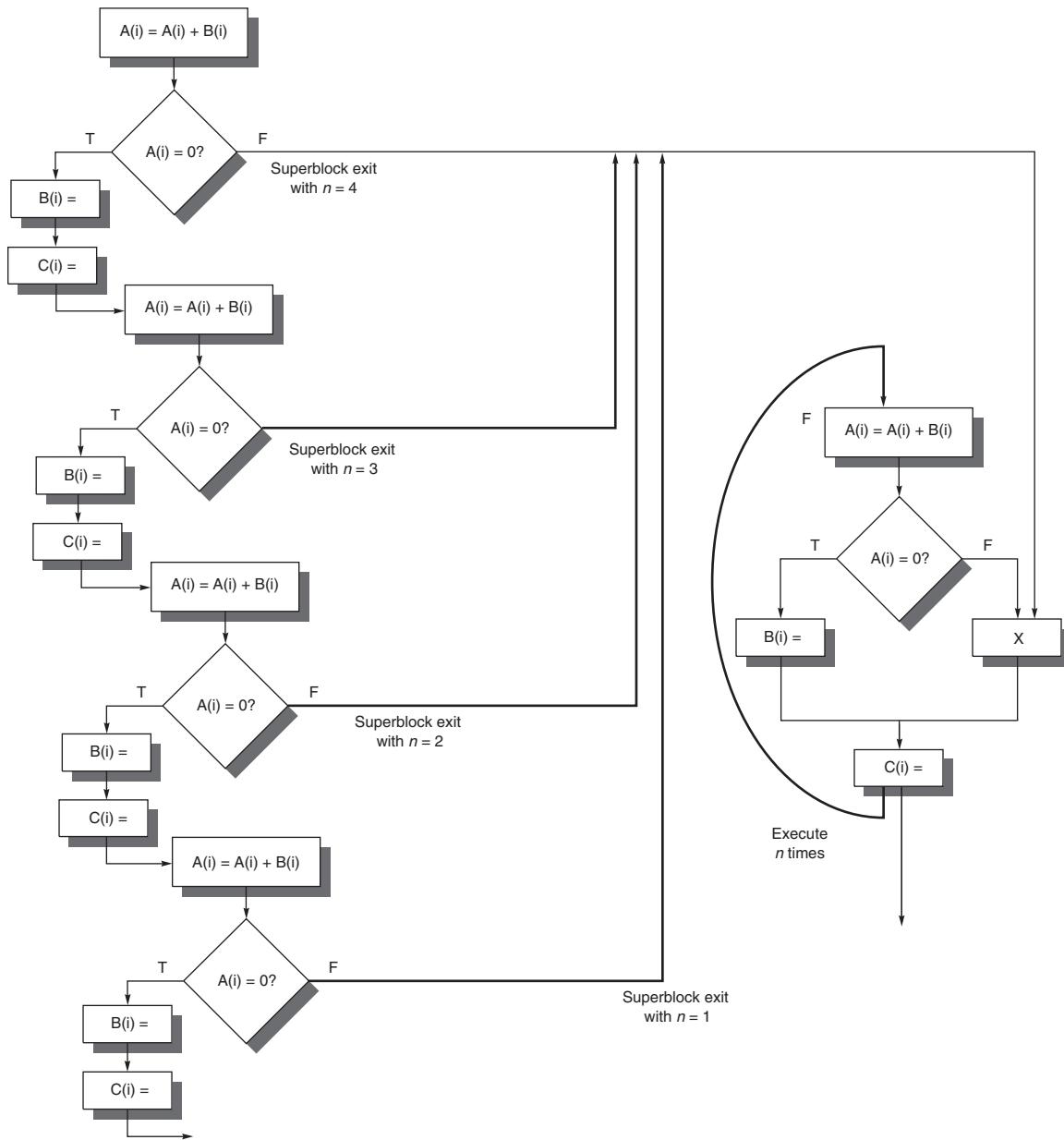


Figure H.5 This superblock results from unrolling the code in Figure H.3 four times and creating a superblock.

each of these techniques and their suitability for various architectural approaches are among the hottest topics being actively pursued by researchers and designers of high-speed processors.

## H.4

### Hardware Support for Exposing Parallelism: Predicated Instructions

Techniques such as loop unrolling, software pipelining, and trace scheduling can be used to increase the amount of parallelism available when the behavior of branches is fairly predictable at compile time. When the behavior of branches is not well known, compiler techniques alone may not be able to uncover much ILP. In such cases, the control dependences may severely limit the amount of parallelism that can be exploited. To overcome these problems, an architect can extend the instruction set to include *conditional* or *predicated instructions*. Such instructions can be used to eliminate branches, converting a control dependence into a data dependence and potentially improving performance. Such approaches are useful with either the hardware-intensive schemes in Chapter 3 or the software-intensive approaches discussed in this appendix, since in both cases predication can be used to eliminate branches.

The concept behind conditional instructions is quite simple: An instruction refers to a condition, which is evaluated as part of the instruction execution. If the condition is true, the instruction is executed normally; if the condition is false, the execution continues as if the instruction were a no-op. Many newer architectures include some form of conditional instructions. The most common example of such an instruction is conditional move, which moves a value from one register to another if the condition is true. Such an instruction can be used to completely eliminate a branch in simple code sequences.

**Example** Consider the following code:

```
if (A==0) { S=T; }
```

Assuming that registers R1, R2, and R3 hold the values of A, S, and T, respectively, show the code for this statement with the branch and with the conditional move.

**Answer** The straightforward code using a branch for this statement is (remember that we are assuming normal rather than delayed branches)

|      |              |
|------|--------------|
| BNEZ | R1 , L       |
| ADDU | R2 , R3 , R0 |
| L:   |              |

Using a conditional move that performs the move only if the third operand is equal to zero, we can implement this statement in one instruction:

```
CMOVZ R2 , R3 , R1
```

The conditional instruction allows us to convert the control dependence present in the branch-based code sequence to a data dependence. (This transformation is also used for vector computers, where it is called *if conversion*.) For a pipelined processor, this moves the place where the dependence must be resolved from near the front of the pipeline, where it is resolved for branches, to the end of the pipeline, where the register write occurs.

---

One obvious use for conditional move is to implement the absolute value function:  $A = \text{abs}(B)$ , which is implemented as `if ( $B < 0$ ) { $A = -B$ ; } else { $A = B$ ; }`. This if statement can be implemented as a pair of conditional moves, or as one unconditional move ( $A = B$ ) and one conditional move ( $A = -B$ ).

In the example above or in the compilation of absolute value, conditional moves are used to change a control dependence into a data dependence. This enables us to eliminate the branch and possibly improve the pipeline behavior. As issue rates increase, designers are faced with one of two choices: execute multiple branches per clock cycle or find a method to eliminate branches to avoid this requirement. Handling multiple branches per clock is complex, since one branch must be control dependent on the other. The difficulty of accurately predicting two branch outcomes, updating the prediction tables, and executing the correct sequence has so far caused most designers to avoid processors that execute multiple branches per clock. Conditional moves and predicated instructions provide a way of reducing the branch pressure. In addition, a conditional move can often eliminate a branch that is hard to predict, increasing the potential gain.

Conditional moves are the simplest form of conditional or predicated instructions and, although useful for short sequences, have limitations. In particular, using conditional move to eliminate branches that guard the execution of large blocks of code can be inefficient, since many conditional moves may need to be introduced.

To remedy the inefficiency of using conditional moves, some architectures support full predication, whereby the execution of all instructions is controlled by a predicate. When the predicate is false, the instruction becomes a no-op. Full predication allows us to simply convert large blocks of code that are branch dependent. For example, an if-then-else statement within a loop can be entirely converted to predicated execution, so that the code in the then case executes only if the value of the condition is true and the code in the else case executes only if the value of the condition is false. Predication is particularly valuable with global code scheduling, since it can eliminate nonloop branches, which significantly complicate instruction scheduling.

Predicated instructions can also be used to speculatively move an instruction that is time critical, but may cause an exception if moved before a guarding branch. Although it is possible to do this with conditional move, it is more costly.

---

**Example** Here is a code sequence for a two-issue superscalar that can issue a combination of one memory reference and one ALU operation, or a branch by itself, every cycle:

| First instruction slot | Second instruction slot |
|------------------------|-------------------------|
| LW                     | R1,40<br>(R2)           |
|                        | ADD R3,R4,R5            |
|                        | ADD R6,R3,R7            |
| BEQZ                   | R10,L                   |
| LW                     | R8,0<br>(R10)           |
| LW                     | R9,0(R8)                |

This sequence wastes a memory operation slot in the second cycle and will incur a data dependence stall if the branch is not taken, since the second LW after the branch depends on the prior load. Show how the code can be improved using a predicated form of LW.

**Answer** Call the predicated version load word LWC and assume the load occurs unless the third operand is 0. The LW immediately following the branch can be converted to an LWC and moved up to the second issue slot:

| First instruction slot | Second instruction slot |
|------------------------|-------------------------|
| LW                     | R1,40(R2)               |
| LWC                    | ADD R3,R4,R5            |
| BEQZ                   | R8,0(R10),R10           |
| LW                     | ADD R6,R3,R7            |
|                        | R10,L                   |
|                        | R9,0(R8)                |

This improves the execution time by several cycles since it eliminates one instruction issue slot and reduces the pipeline stall for the last instruction in the sequence. Of course, if the compiler mispredicted the branch, the predicated instruction will have no effect and will not improve the running time. This is why the transformation is speculative.

If the sequence following the branch were short, the entire block of code might be converted to predicated execution and the branch eliminated.

---

When we convert an entire code segment to predicated execution or speculatively move an instruction and make it predicted, we remove a control dependence. Correct code generation and the conditional execution of predicated instructions ensure that we maintain the data flow enforced by the branch. To ensure that the exception behavior is also maintained, a predicated instruction must not generate an exception if the predicate is false. The property of not causing exceptions is

quite critical, as the previous example shows: If register R10 contains zero, the instruction LW R8,0(R10) executed unconditionally is likely to cause a protection exception, and this exception should not occur. Of course, if the condition is satisfied (i.e., R10 is not zero), the LW may still cause a legal and resumable exception (e.g., a page fault), and the hardware must take the exception when it knows that the controlling condition is true.

The major complication in implementing predicated instructions is deciding when to annul an instruction. Predicated instructions may either be annulled during instruction issue or later in the pipeline before they commit any results or raise an exception. Each choice has a disadvantage. If predicated instructions are annulled early in the pipeline, the value of the controlling condition must be known early to prevent a stall for a data hazard. Since data-dependent branch conditions, which tend to be less predictable, are candidates for conversion to predicated execution, this choice can lead to more pipeline stalls. Because of this potential for data hazard stalls, no design with predicated execution (or conditional move) annuls instructions early. Instead, all existing processors annul instructions later in the pipeline, which means that annulled instructions will consume functional unit resources and potentially have a negative impact on performance. A variety of other pipeline implementation techniques, such as forwarding, interact with predicated instructions, further complicating the implementation.

Predicated or conditional instructions are extremely useful for implementing short alternative control flows, for eliminating some unpredictable branches, and for reducing the overhead of global code scheduling. Nonetheless, the usefulness of conditional instructions is limited by several factors:

- Predicated instructions that are annulled (i.e., whose conditions are false) still take some processor resources. An annulled predicated instruction requires fetch resources at a minimum, and in most processors functional unit execution time. Therefore, moving an instruction across a branch and making it conditional will slow the program down whenever the moved instruction would not have been normally executed. Likewise, predicing a control-dependent portion of code and eliminating a branch may slow down the processor if that code would not have been executed. An important exception to these situations occurs when the cycles used by the moved instruction when it is not performed would have been idle anyway (as in the earlier superscalar example). Moving an instruction across a branch or converting a code segment to predicated execution is essentially speculating on the outcome of the branch. Conditional instructions make this easier but do not eliminate the execution time taken by an incorrect guess. In simple cases, where we trade a conditional move for a branch and a move, using conditional moves or predication is almost always better. When longer code sequences are made conditional, the benefits are more limited.
- Predicated instructions are most useful when the predicate can be evaluated early. If the condition evaluation and predicated instructions cannot be

separated (because of data dependences in determining the condition), then a conditional instruction may result in a stall for a data hazard. With branch prediction and speculation, such stalls can be avoided, at least when the branches are predicted accurately.

- The use of conditional instructions can be limited when the control flow involves more than a simple alternative sequence. For example, moving an instruction across multiple branches requires making it conditional on both branches, which requires two conditions to be specified or requires additional instructions to compute the controlling predicate. If such capabilities are not present, the overhead of if conversion will be larger, reducing its advantage.
- Conditional instructions may have some speed penalty compared with unconditional instructions. This may show up as a higher cycle count for such instructions or a slower clock rate overall. If conditional instructions are more expensive, they will need to be used judiciously.

For these reasons, many architectures have included a few simple conditional instructions (with conditional move being the most frequent), but only a few architectures include conditional versions for the majority of the instructions. The MIPS, Alpha, PowerPC, SPARC, and Intel x86 (as defined in the Pentium processor) all support conditional move. The IA-64 architecture supports full predication for all instructions, as we will see in Section H.6.

## H.5

### Hardware Support for Compiler Speculation

As we saw in Chapter 3, many programs have branches that can be accurately predicted at compile time either from the program structure or by using a profile. In such cases, the compiler may want to speculate either to improve the scheduling or to increase the issue rate. Predicated instructions provide one method to speculate, but they are really more useful when control dependences can be completely eliminated by if conversion. In many cases, we would like to move speculated instructions not only before the branch but also before the condition evaluation, and predication cannot achieve this.

To speculate ambitiously requires three capabilities:

1. The ability of the compiler to find instructions that, with the possible use of register renaming, can be speculatively moved and not affect the program data flow
2. The ability to ignore exceptions in speculated instructions, until we know that such exceptions should really occur
3. The ability to speculatively interchange loads and stores, or stores and stores, which may have address conflicts

The first of these is a compiler capability, while the last two require hardware support, which we explore next.

## Hardware Support for Preserving Exception Behavior

To speculate ambitiously, we must be able to move any type of instruction and still preserve its exception behavior. The key to being able to do this is to observe that the results of a speculated sequence that is mispredicted will not be used in the final computation, and such a speculated instruction should not cause an exception.

There are four methods that have been investigated for supporting more ambitious speculation without introducing erroneous exception behavior:

1. The hardware and operating system cooperatively ignore exceptions for speculative instructions. As we will see later, this approach preserves exception behavior for correct programs, but not for incorrect ones. This approach may be viewed as unacceptable for some programs, but it has been used, under program control, as a “fast mode” in several processors.
2. Speculative instructions that never raise exceptions are used, and checks are introduced to determine when an exception should occur.
3. A set of status bits, called *poison bits*, are attached to the result registers written by speculated instructions when the instructions cause exceptions. The poison bits cause a fault when a normal instruction attempts to use the register.
4. A mechanism is provided to indicate that an instruction is speculative, and the hardware buffers the instruction result until it is certain that the instruction is no longer speculative.

To explain these schemes, we need to distinguish between exceptions that indicate a program error and would normally cause termination, such as a memory protection violation, and those that are handled and normally resumed, such as a page fault. Exceptions that can be resumed can be accepted and processed for speculative instructions just as if they were normal instructions. If the speculative instruction should not have been executed, handling the unneeded exception may have some negative performance effects, but it cannot cause incorrect execution. The cost of these exceptions may be high, however, and some processors use hardware support to avoid taking such exceptions, just as processors with hardware speculation may take some exceptions in speculative mode, while avoiding others until an instruction is known not to be speculative.

Exceptions that indicate a program error should not occur in correct programs, and the result of a program that gets such an exception is not well defined, except perhaps when the program is running in a debugging mode. If such exceptions arise in speculated instructions, we cannot take the exception until we know that the instruction is no longer speculative.

In the simplest method for preserving exceptions, the hardware and the operating system simply handle all resumable exceptions when the exception occurs and simply return an undefined value for any exception that would cause termination. If the instruction generating the terminating exception was not speculative, then the program is in error. Note that instead of terminating the program, the

program is allowed to continue, although it will almost certainly generate incorrect results. If the instruction generating the terminating exception is speculative, then the program may be correct and the speculative result will simply be unused; thus, returning an undefined value for the instruction cannot be harmful. This scheme can never cause a correct program to fail, no matter how much speculation is done. An incorrect program, which formerly might have received a terminating exception, will get an incorrect result. This is acceptable for some programs, assuming the compiler can also generate a normal version of the program, which does not speculate and can receive a terminating exception.

**Example** Consider that the following code fragment from an if-then-else statement of the form

```
if (A==0) A = B; else A = A+4;
```

where A is at 0(R3) and B is at 0(R2):

|      |                |              |
|------|----------------|--------------|
| LD   | R1,0(R3)       | ;load A      |
| BNEZ | R1,L1          | ;test A      |
| LD   | R1,0(R2)       | ;then clause |
| J    | L2             | ;skip else   |
| L1:  | DADDI R1,R1,#4 | ;else clause |
| L2:  | SD R1,0(R3)    | ;store A     |

Assume that the then clause is *almost always* executed. Compile the code using compiler-based speculation. Assume R14 is unused and available.

**Answer** Here is the new code:

|       |              |                         |
|-------|--------------|-------------------------|
| LD    | R1,0(R3)     | ;load A                 |
| LD    | R14,0(R2)    | ;speculative load B     |
| BEQZ  | R1,L3        | ;other branch of the if |
| DADDI | R14,R1,#4    | ;the else clause        |
| L3:   | SD R14,0(R3) | ;nonspeculative store   |

The then clause is completely speculated. We introduce a temporary register to avoid destroying R1 when B is loaded; if the load is speculative, R14 will be useless. After the entire code segment is executed, A will be in R14. The else clause could have also been compiled speculatively with a conditional move, but if the branch is highly predictable and low cost, this might slow the code down, since two extra instructions would always be executed as opposed to one branch.

In such a scheme, it is not necessary to know that an instruction is speculative. Indeed, it is helpful only when a program is in error and receives a terminating exception on a normal instruction; in such cases, if the instruction were not marked as speculative, the program could be terminated.

In this method for handling speculation, as in the next one, renaming will often be needed to prevent speculative instructions from destroying live values. Renaming is usually restricted to register values. Because of this restriction, the targets of stores cannot be destroyed and stores cannot be speculative. The small number of registers and the cost of spilling will act as one constraint on the amount of speculation. Of course, the major constraint remains the cost of executing speculative instructions when the compiler's branch prediction is incorrect.

A second approach to preserving exception behavior when speculating introduces speculative versions of instructions that do not generate terminating exceptions and instructions to check for such exceptions. This combination preserves the exception behavior exactly.

**Example** Show how the previous example can be coded using a speculative load (SLD) and a speculation check instruction (SPECCK) to completely preserve exception behavior. Assume R14 is unused and available.

**Answer** Here is the code that achieves this:

```

LD R1,0(R3) ;load A
sLD R14,0(R2) ;speculative, no termination
BNEZ R1,L1 ;test A
SPECCK 0(R2) ;perform speculation check
J L2 ;skip else
L1: DADDI R14,R1,#4 ;else clause
L2: SD R14,0(R3) ;store A

```

Notice that the speculation check requires that we maintain a basic block for the then case. If we had speculated only a portion of the then case, then a basic block representing the then case would exist in any event. More importantly, notice that checking for a possible exception requires extra code.

A third approach for preserving exception behavior tracks exceptions as they occur but postpones any terminating exception until a value is actually used, preserving the occurrence of the exception, although not in a completely precise fashion. The scheme is simple: A poison bit is added to every register, and another bit is added to every instruction to indicate whether the instruction is speculative. The poison bit of the destination register is set whenever a speculative instruction results in a terminating exception; all other exceptions are handled immediately. If a speculative instruction uses a register with a poison bit turned on, the destination register of the instruction simply has its poison bit turned on. If a normal instruction attempts to use a register source with its poison bit turned on, the instruction causes a fault. In this way, any program that would have generated an exception still generates one, albeit at the first instance where a result is used by an instruction that is not speculative. Since poison bits exist only on register

values and not memory values, stores are never speculative and thus trap if either operand is “poison.”

---

**Example** Consider the code fragment from page H-29 and show how it would be compiled with speculative instructions and poison bits. Show where an exception for the speculative memory reference would be recognized. Assume R14 is unused and available.

**Answer** Here is the code (an s preceding the opcode indicates a speculative instruction):

```

LD R1,0(R3) ;load A
sLD R14,0(R2) ;speculative load B
BEQZ R1,L3 ;
DADDI R14,R1,#4 ;
L3: SD R14,0(R3) ;exception for speculative LW

```

If the speculative s LD generates a terminating exception, the poison bit of R14 will be turned on. When the nonspeculative SW instruction occurs, it will raise an exception if the poison bit for R14 is on.

---

One complication that must be overcome is how the OS saves the user registers on a context switch if the poison bit is set. A special instruction is needed to save and reset the state of the poison bits to avoid this problem.

The fourth and final approach listed earlier relies on a hardware mechanism that operates like a reorder buffer. In such an approach, instructions are marked by the compiler as speculative and include an indicator of how many branches the instruction was speculatively moved across and what branch action (taken/not taken) the compiler assumed. This last piece of information basically tells the hardware the location of the code block where the speculated instruction originally was. In practice, most of the benefit of speculation is gained by allowing movement across a single branch; thus, only 1 bit saying whether the speculated instruction came from the taken or not taken path is required. Alternatively, the original location of the speculative instruction is marked by a *sentinel*, which tells the hardware that the earlier speculative instruction is no longer speculative and values may be committed.

All instructions are placed in a reorder buffer when issued and are forced to commit in order, as in a hardware speculation approach. (Notice, though, that no actual speculative branch prediction or dynamic scheduling occurs.) The reorder buffer tracks when instructions are ready to commit and delays the “write-back” portion of any speculative instruction. Speculative instructions are not allowed to commit until the branches that have been speculatively moved over are also ready to commit, or, alternatively, until the corresponding sentinel is reached. At that point, we know whether the speculated instruction should have been executed or not. If it should have been executed and it generated a terminating

exception, then we know that the program should be terminated. If the instruction should not have been executed, then the exception can be ignored. Notice that the compiler, rather than the hardware, has the job of register renaming to ensure correct usage of the speculated result, as well as correct program execution.

### Hardware Support for Memory Reference Speculation

Moving loads across stores is usually done when the compiler is certain the addresses do not conflict. As we saw with the examples in Section 3.2, such transformations are critical to reducing the critical path length of a code segment. To allow the compiler to undertake such code motion when it cannot be absolutely certain that such a movement is correct, a special instruction to check for address conflicts can be included in the architecture. The special instruction is left at the original location of the load instruction (and acts like a guardian), and the load is moved up across one or more stores.

When a speculated load is executed, the hardware saves the address of the accessed memory location. If a subsequent store changes the location before the check instruction, then the speculation has failed. If the location has not been touched, then the speculation is successful. Speculation failure can be handled in two ways. If only the load instruction was speculated, then it suffices to redo the load at the point of the check instruction (which could supply the target register in addition to the memory address). If additional instructions that depended on the load were also speculated, then a fix-up sequence that reexecutes all the speculated instructions starting with the load is needed. In this case, the check instruction specifies the address where the fix-up code is located.

In this section, we have seen a variety of hardware assist mechanisms. Such mechanisms are key to achieving good support with the compiler-intensive approaches of Chapter 3 and this appendix. In addition, several of them can be easily integrated in the hardware-intensive approaches of Chapter 3 and provide additional benefits.

## H.6

### The Intel IA-64 Architecture and Itanium Processor

This section is an overview of the Intel IA-64 architecture, the most advanced VLIW-style processor, and its implementation in the Itanium processor.

#### The Intel IA-64 Instruction Set Architecture

The IA-64 is a RISC-style, register-register instruction set, but with many novel features designed to support compiler-based exploitation of ILP. Our focus here is on the unique aspects of the IA-64 ISA. Most of these aspects have been discussed already in this appendix, including predication, compiler-based parallelism detection, and support for memory reference speculation.

When they announced the IA-64 architecture, HP and Intel introduced the term EPIC (Explicitly Parallel Instruction Computer) to distinguish this new architectural approach from the earlier VLIW architectures and from other RISC architectures. Although VLIW and EPIC architectures share many features, the EPIC approach includes several concepts that extend the earlier VLIW approach. These extensions fall into two main areas:

1. EPIC has greater flexibility in indicating parallelism among instructions and in instruction formats. Rather than relying on a fixed instruction format where all operations in the instruction must be capable of being executed in parallel and where the format is completely rigid, EPIC uses explicit indicators of possible instruction dependence as well as a variety of instruction formats. This EPIC approach can express parallelism more flexibly than the more rigid VLIW method and can reduce the increases in code size caused by the typically inflexible VLIW instruction format.
2. EPIC has more extensive support for software speculation than the earlier VLIW schemes that had only minimal support.

In addition, the IA-64 architecture includes a variety of features to improve performance, such as register windows and a rotating floating-point register (FPR) stack.

### *The IA-64 Register Model*

The components of the IA-64 register state are

- 128 64-bit general-purpose registers, which as we will see shortly are actually 65 bits wide
- 128 82-bit floating-point registers, which provide two extra exponent bits over the standard 80-bit IEEE format
- 64 1-bit predicate registers
- 8 64-bit branch registers, which are used for indirect branches
- A variety of registers used for system control, memory mapping, performance counters, and communication with the OS

The integer registers are configured to help accelerate procedure calls using a register stack mechanism similar to that developed in the Berkeley RISC-I processor and used in the SPARC architecture. Registers 0 to 31 are always accessible and are addressed as 0 to 31. Registers 32 to 128 are used as a register stack, and each procedure is allocated a set of registers (from 0 to 96) for its use. The new register stack frame is created for a called procedure by renaming the registers in hardware; a special register called the current frame pointer (CFM) points to the set of registers to be used by a given procedure. The frame consists of two parts: the local area and the output area. The local area is used for local storage, while

the output area is used to pass values to any called procedure. The `alloc` instruction specifies the size of these areas. Only the integer registers have register stack support.

On a procedure call, the CFM pointer is updated so that R32 of the called procedure points to the first register of the output area of the called procedure. This update enables the parameters of the caller to be passed into the addressable registers of the callee. The callee executes an `alloc` instruction to allocate both the number of required local registers, which include the output registers of the caller, and the number of output registers needed for parameter passing to a called procedure. Special load and store instructions are available for saving and restoring the register stack, and special hardware (called the *register stack engine*) handles overflow of the register stack.

In addition to the integer registers, there are three other sets of registers: the floating-point registers, the predicate registers, and the branch registers. The floating-point registers are used for floating-point data, and the branch registers are used to hold branch destination addresses for indirect branches. The predication registers hold predicates, which control the execution of predicated instructions; we describe the predication mechanism later in this section.

Both the integer and floating-point registers support register rotation for registers 32 to 128. Register rotation is designed to ease the task of allocating registers in software-pipelined loops, a problem that we discussed in Section H.3. In addition, when combined with the use of predication, it is possible to avoid the need for unrolling and for separate prologue and epilogue code for a software-pipelined loop. This capability reduces the code expansion incurred to use software pipelining and makes the technique usable for loops with smaller numbers of iterations, where the overheads would traditionally negate many of the advantages.

#### *Instruction Format and Support for Explicit Parallelism*

The IA-64 architecture is designed to achieve the major benefits of a VLIW approach—implicit parallelism among operations in an instruction and fixed formatting of the operation fields—while maintaining greater flexibility than a VLIW normally allows. This combination is achieved by relying on the compiler to detect ILP and schedule instructions into parallel instruction slots, but adding flexibility in the formatting of instructions and allowing the compiler to indicate when an instruction cannot be executed in parallel with its successors.

The IA-64 architecture uses two different concepts to achieve the benefits of implicit parallelism and ease of instruction decode. Implicit parallelism is achieved by placing instructions into *instruction groups*, while the fixed formatting of multiple instructions is achieved through the introduction of a concept called a *bundle*, which contains three instructions. Let's start by defining an instruction group.

An instruction group is a sequence of consecutive instructions with no register data dependences among them (there are a few minor exceptions). All the instructions in a group could be executed in parallel, if sufficient hardware resources existed and if any dependences through memory were preserved. An instruction group can be arbitrarily long, but the compiler must *explicitly* indicate the

| Execution unit slot | Instruction type | Instruction description | Example instructions                                    |
|---------------------|------------------|-------------------------|---------------------------------------------------------|
| I-unit              | A                | Integer ALU             | Add, subtract, and, or, compare                         |
|                     | I                | Non-ALU                 | integer Integer and multimedia shifts, bit tests, moves |
| M-unit              | A                | Integer ALU             | Add, subtract, and, or, compare                         |
|                     | M                | Memory access           | Loads and stores for integer/FP registers               |
| F-unit              | F                | Floating point          | Floating-point instructions                             |
| B-unit              | B                | Branches                | Conditional branches, calls, loop branches              |
| L + X               | L + X            | Extended                | Extended immediates, stops and no-ops                   |

**Figure H.6** The five execution unit slots in the IA-64 architecture and what instruction types they may hold are shown. A-type instructions, which correspond to integer ALU instructions, may be placed in either an I-unit or M-unit slot. L + X slots are special, as they occupy two instruction slots; L + X instructions are used to encode 64-bit immediates and a few special instructions. L + X instructions are executed either by the I-unit or the B-unit.

boundary between one instruction group and another. This boundary is indicated by placing a *stop* between two instructions that belong to different groups. To understand how stops are indicated, we must first explain how instructions are placed into bundles.

IA-64 instructions are encoded in bundles, which are 128 bits wide. Each bundle consists of a 5-bit template field and three instructions, each 41 bits in length. (Actually, the 41-bit quantities are not truly instructions, since they can only be interpreted in conjunction with the template field. The name *syllable* is sometimes used for these operations. For simplicity, we will continue to use the term “instruction.”) To simplify the decoding and instruction issue process, the template field of a bundle specifies what types of execution units each instruction in the bundle requires. Figure H.6 shows the five different execution unit types and describes what instruction classes they may hold, together with some examples.

The 5-bit template field within each bundle describes *both* the presence of any stops associated with the bundle and the execution unit type required by each instruction within the bundle. Figure H.7 shows the possible formats that the template field encodes and the position of any stops it specifies. The bundle formats can specify only a subset of all possible combinations of instruction types and stops. To see how the bundle works, let’s consider an example.

---

**Example** Unroll the array increment example,  $x[i] = x[i] + s$ , seven times and place the instructions into bundles, first ignoring pipeline latencies (to minimize the number of bundles) and then scheduling the code to minimize stalls. In scheduling the code assume one bundle executes per clock and that any stalls cause the entire bundle to

| Template | Slot 0 | Slot 1 | Slot 2 |
|----------|--------|--------|--------|
| 0        | M      | I      | I      |
| 1        | M      | I      | I      |
| 2        | M      | I      | I      |
| 3        | M      | I      | I      |
| 4        | M      | L      | X      |
| 5        | M      | L      | X      |
| 8        | M      | M      | I      |
| 9        | M      | M      | I      |
| 10       | M      | M      | I      |
| 11       | M      | M      | I      |
| 12       | M      | F      | I      |
| 13       | M      | F      | I      |
| 14       | M      | M      | F      |
| 15       | M      | M      | F      |
| 16       | M      | I      | B      |
| 17       | M      | I      | B      |
| 18       | M      | B      | B      |
| 19       | M      | B      | B      |
| 22       | B      | B      | B      |
| 23       | B      | B      | B      |
| 24       | M      | M      | B      |
| 25       | M      | M      | B      |
| 28       | M      | F      | B      |
| 29       | M      | F      | B      |

**Figure H.7** The 24 possible template values (8 possible values are reserved) and the instruction slots and stops for each format. Stops are indicated by heavy lines and may appear within and/or at the end of the bundle. For example, template 9 specifies that the instruction slots are M, M, and I (in that order) and that the only stop is between this bundle and the next. Template 11 has the same type of instruction slots but also includes a stop after the first slot. The L + X format is used when slot 1 is L and slot 2 is X.

be stalled. Use the pipeline latencies from Figure 3.2. Use MIPS instruction mnemonics for simplicity.

**Answer** The two different versions are shown in Figure H.8. Although the latencies are different from those in Itanium, the most common bundle, MMF, must be issued by itself in Itanium, just as our example assumes.

| Bundle template                                          | Slot 0          | Slot 1            | Slot 2           | Execute cycle<br>(1 bundle/<br>cycle) |
|----------------------------------------------------------|-----------------|-------------------|------------------|---------------------------------------|
| 9: M M I                                                 | L.D F0,0(R1)    | L.D F6,-8(R1)     |                  | 1                                     |
| 14: M M F                                                | L.D F10,-16(R1) | L.D F14,-24(R1)   | ADD.D F4,F0,F2   | 3                                     |
| 15: M M F                                                | L.D F18,-32(R1) | L.D F22,-40(R1)   | ADD.D F8,F6,F2   | 4                                     |
| 15: M M F                                                | L.D F26,-48(R1) | S.D F4,0(R1)      | ADD.D F12,F10,F2 | 6                                     |
| 15: M M F                                                | S.D F8,-8(R1)   | S.D F12,-16(R1)   | ADD.D F16,F14,F2 | 9                                     |
| 15: M M F                                                | S.D F16,-24(R1) |                   | ADD.D F20,F18,F2 | 12                                    |
| 15: M M F                                                | S.D F20,-32(R1) |                   | ADD.D F24,F22,F2 | 15                                    |
| 15: M M F                                                | S.D F24,-40(R1) |                   | ADD.D F28,F26,F2 | 18                                    |
| 16: M I B                                                | S.D F28,-48(R1) | DADDUI R1,R1,#-56 | BNE R1,R2,Loop   | 21                                    |
| (a) The code scheduled to minimize the number of bundles |                 |                   |                  |                                       |

| Bundle template                                                                                | Slot 0          | Slot 1            | Slot 2           | Execute cycle<br>(1 bundle/<br>cycle) |
|------------------------------------------------------------------------------------------------|-----------------|-------------------|------------------|---------------------------------------|
| 8: M M I                                                                                       | L.D F0,0(R1)    | L.D F6,-8(R1)     |                  | 1                                     |
| 9: M M I                                                                                       | L.D F10,-16(R1) | L.D F14,-24(R1)   |                  | 2                                     |
| 14: M M F                                                                                      | L.D F18,-32(R1) | L.D F22,-40(R1)   | ADD.D F4,F0,F2   | 3                                     |
| 14: M M F                                                                                      | L.D F26,-48(R1) |                   | ADD.D F8,F6,F2   | 4                                     |
| 15: M M F                                                                                      |                 |                   | ADD.D F12,F10,F2 | 5                                     |
| 14: M M F                                                                                      |                 | S.D F4,0(R1)      | ADD.D F16,F14,F2 | 6                                     |
| 14: M M F                                                                                      |                 | S.D F8,-8(R1)     | ADD.D F20,F18,F2 | 7                                     |
| 15: M M F                                                                                      |                 | S.D F12,-16(R1)   | ADD.D F24,F22,F2 | 8                                     |
| 14: M M F                                                                                      |                 | S.D F16,-24(R1)   | ADD.D F28,F26,F2 | 9                                     |
| 9: M M I                                                                                       | S.D F20,-32(R1) | S.D F24,-40(R1)   |                  | 11                                    |
| 16: M I B                                                                                      | S.D F28,-48(R1) | DADDUI R1,R1,#-56 | BNE R1,R2,Loop   | 12                                    |
| (b) The code scheduled to minimize the number of cycles assuming one bundle executed per cycle |                 |                   |                  |                                       |

**Figure H.8** The IA-64 instructions, including bundle bits and stops, for the unrolled version of  $x[i] = x[i] + s$ , when unrolled seven times and scheduled (a) to minimize the number of instruction bundles and (b) to minimize the number of cycles (assuming that a hazard stalls an entire bundle). Blank entries indicate unused slots, which are encoded as no-ops. The absence of stops indicates that some bundles could be executed in parallel. Minimizing the number of bundles yields 9 bundles versus the 11 needed to minimize the number of cycles. The scheduled version executes in just over half the number of cycles. Version (a) fills 85% of the instruction slots, while (b) fills 70%. The number of empty slots in the scheduled code and the use of bundles may lead to code sizes that are much larger than other RISC architectures. Note that the branch in the last bundle in both sequences depends on the DADD in the same bundle. In the IA-64 instruction set, this sequence would be coded as a setting of a predication register and a branch that would be predicated on that register. Normally, such dependent operations could not occur in the same bundle, but this case is one of the exceptions mentioned earlier.

### *Instruction Set Basics*

Before turning to the special support for speculation, we briefly discuss the major instruction encodings and survey the instructions in each of the five primary instruction classes (A, I, M, F, and B). Each IA-64 instruction is 41 bits in length. The high-order 4 bits, together with the bundle bits that specify the execution unit slot, are used as the major opcode. (That is, the 4-bit opcode field is reused across the execution field slots, and it is appropriate to think of the opcode as being 4 bits plus the M, F, I, B, L + X designation.) The low-order 6 bits of every instruction are used for specifying the predicate register that guards the instruction (see the next subsection).

Figure H.9 summarizes most of the major instruction formats, other than the multimedia instructions, and gives examples of the instructions encoded for each format.

### *Predication and Speculation Support*

The IA-64 architecture provides comprehensive support for predication: Nearly every instruction in the IA-64 architecture can be predicated. An instruction is predicated by specifying a predicate register, whose identity is placed in the lower 6 bits of each instruction field. Because nearly all instructions can be predicated, both if conversion and code motion have lower overhead than they would with only limited support for conditional instructions. One consequence of full predication is that a conditional branch is simply a branch with a guarding predicate!

Predicate registers are set using compare or test instructions. A compare instruction specifies one of ten different comparison tests and two predicate registers as destinations. The two predicate registers are written either with the result of the comparison (0 or 1) and the complement, or with some logical function that combines the two tests (such as *and*) and the complement. This capability allows multiple comparisons to be done in one instruction.

Speculation support in the IA-64 architecture consists of separate support for control speculation, which deals with deferring exception for speculated instructions, and memory reference speculation, which supports speculation of load instructions.

Deferred exception handling for speculative instructions is supported by providing the equivalent of poison bits. For the general-purpose registers (GPRs), these bits are called NaTs (Not a Thing), and this extra bit makes the GPRs effectively 65 bits wide. For the FP registers this capability is obtained using a special value, NaTVal (Not a Thing Value); this value is encoded using a significand of 0 and an exponent outside of the IEEE range. Only speculative load instructions generate such values, but all instructions that do not affect memory will cause a NaT or NaTVal to be propagated to the result register. (There are both speculative and non-speculative loads; the latter can only raise immediate exceptions and cannot defer them.) Floating-point exceptions are not handled through this mechanism but instead use floating-point status registers to record exceptions.

| Instruction type | Number of formats | Representative instructions                                                  | Extra opcode bits | GPRs/ FPRs | Immediate bits        | Other/comment                                 |
|------------------|-------------------|------------------------------------------------------------------------------|-------------------|------------|-----------------------|-----------------------------------------------|
| A                | 8                 | Add, subtract, and, or                                                       | 9                 | 3          | 0                     |                                               |
|                  |                   | Shift left and add                                                           | 7                 | 3          | 0                     | 2-bit shift count                             |
|                  |                   | ALU immediates                                                               | 9                 | 2          | 8                     |                                               |
|                  |                   | Add immediate                                                                | 3                 | 2          | 14                    |                                               |
|                  |                   | Add immediate                                                                | 0                 | 2          | 22                    |                                               |
|                  |                   | Compare                                                                      | 4                 | 2          | 0                     | 2 predicate register destinations             |
|                  |                   | Compare immediate                                                            | 3                 | 1          | 8                     | 2 predicate register destinations             |
| I                | 29                | Shift R/L variable                                                           | 9                 | 3          | 0                     | Many multimedia instructions use this format. |
|                  |                   | Test bit                                                                     | 6                 | 3          | 6-bit field specifier | 2 predicate register destinations             |
|                  |                   | Move to BR                                                                   | 6                 | 1          | 9-bit branch predict  | Branch register specifier                     |
| M                | 46                | Integer/FP load and store, line prefetch                                     | 10                | 2          | 0                     | Speculative/nonspeculative                    |
|                  |                   | Integer/FP load and store, and line prefetch and post-increment by immediate | 9                 | 2          | 8                     | Speculative/nonspeculative                    |
|                  |                   | Integer/FP load prefetch and register postincrement                          | 10                | 3          |                       | Speculative/nonspeculative                    |
|                  |                   | Integer/FP speculation check                                                 | 3                 | 1          | 21 in two fields      |                                               |
| B                | 9                 | PC-relative branch, counted branch                                           | 7                 | 0          | 21                    |                                               |
|                  |                   | PC-relative call                                                             | 4                 | 0          | 21                    | 1 branch register                             |
| F                | 15                | FP arithmetic                                                                | 2                 | 4          |                       |                                               |
|                  |                   | FP compare                                                                   | 2                 | 2          |                       | 2 6-bit predicate regs                        |
| L + X            | 4                 | Move immediate long                                                          | 2                 | 1          | 64                    |                                               |

**Figure H.9 A summary of some of the instruction formats of the IA-64 ISA.** The major opcode bits and the guarding predication register specifier add 10 bits to every instruction. The number of formats indicated for each instruction class in the second column (a total of 111) is a strict interpretation: A different use of a field, even of the same size, is considered a different format. The number of formats that actually have *different field sizes* is one-third to one-half as large. Some instructions have unused bits that are reserved; we have not included those in this table. Immediate bits include the sign bit. The branch instructions include prediction bits, which are used when the predictor does not have a valid prediction. Only one of the many formats for the multimedia instructions is shown in this table.

A deferred exception can be resolved in two different ways. First, if a non-speculative instruction, such as a store, receives a NaT or NaTVal as a source operand, it generates an immediate and unrecoverable exception. Alternatively, a `chk.s` instruction can be used to detect the presence of NaT or NaTVal and branch to a routine designed by the compiler to recover from the speculative operation. Such a recovery approach makes more sense for memory reference speculation.

The inability to store the contents of instructions with a NaT or NaTVal set would make it impossible for the OS to save the state of the processor. Thus, IA-64 includes special instructions to save and restore registers that do not cause an exception for a NaT or NaTVal and also save and restore the NaT bits.

Memory reference support in the IA-64 uses a concept called *advanced loads*. An advanced load is a load that has been speculatively moved above store instructions on which it is potentially dependent. To speculatively perform a load, the `ld.a` (for advanced load) instruction is used. Executing this instruction creates an entry in a special table, called the *ALAT*. The ALAT stores both the register destination of the load and the address of the accessed memory location. When a store is executed, an associative lookup against the active ALAT entries is performed. If there is an ALAT entry with the same memory address as the store, the ALAT entry is marked as invalid.

Before any nonspeculative instruction (i.e., a store) uses the value generated by an advanced load or a value derived from the result of an advanced load, an explicit check is required. The check specifies the destination register of the advanced load. If the ALAT for that register is still valid, the speculation was legal and the only effect of the check is to clear the ALAT entry. If the check fails, the action taken depends on which of two different types of checks was employed. The first type of check is an instruction `ld.c`, which simply causes the data to be reloaded from memory at that point. An `ld.c` instruction is used when *only* the load is advanced. The alternative form of a check, `chk.a`, specifies the address of a fix-up routine that is used to reexecute the load *and any other* speculated code that depended on the value of the load.

## The Itanium 2 Processor

The Itanium 2 processor is the second implementation of the IA-64 architecture. The first version, Itanium 1, became available in 2001 with an 800 MHz clock. The Itanium 2, first delivered in 2003, had a maximum clock rate in 2005 of 1.6 GHz. The two processors are very similar, with some differences in the pipeline structure and greater differences in the memory hierarchies. The Itanium 2 is about four times faster than the Itanium 1. This performance improvement comes from a doubling of the clock rate, a more aggressive memory hierarchy, additional functional units that improve instruction throughput, more complete bypassing, a

shorter pipeline that reduces some stalls, and a more mature compiler system. During roughly the same period that elapsed from the Itanium 1 to Itanium 2, the Pentium processors improved by slightly more than a factor of three. The greater improvement for the Itanium is reasonable given the novelty of the architecture and software system versus the more established IA-32 implementations.

The Itanium 2 can fetch and issue two bundles, or up to six instructions, per clock. The Itanium 2 uses a three-level memory hierarchy all on-chip. The first level uses split instruction and data caches, each 16 KB; floating-point data are not placed in the first-level cache. The second and third levels are unified caches of 256 KB and of 3 MB to 9 MB, respectively.

#### *Functional Units and Instruction Issue*

There are 11 functional units in the Itanium 2 processor: two I-units, four M-units (two for loads and two for stores), three B-units, and two F-units. All the functional units are pipelined. Figure H.10 gives the pipeline latencies for some typical instructions. In addition, when a result is bypassed from one unit to another, there is usually at least one additional cycle of delay.

Itanium 2 can issue up to six instructions per clock from two bundles. In the worst case, if a bundle is split when it is issued, the hardware could see as few as four instructions: one from the first bundle to be executed and three from the second bundle. Instructions are allocated to functional units based on the bundle bits, ignoring the presence of no-ops or predicated instructions with untrue predicates. In addition, when issue to a functional unit is blocked because the next instruction to be issued needs an already committed unit, the resulting bundle is split. A split bundle still occupies one of the two bundle slots, even if it has only one instruction remaining.

| Instruction                      | Latency |
|----------------------------------|---------|
| Integer load                     | 1       |
| Floating-point load              | 5–9     |
| Correctly predicted taken branch | 0–3     |
| Mispredicted branch              | 6       |
| Integer ALU operations           | 0       |
| FP arithmetic                    | 4       |

**Figure H.10** The latency of some typical instructions on Itanium 2. The latency is defined as the smallest number of intervening instructions between two dependent instructions. Integer load latency assumes a hit in the first-level cache. FP loads always bypass the primary cache, so the latency is equal to the access time of the second-level cache. There are some minor restrictions for some of the functional units, but these primarily involve the execution of infrequent instructions.

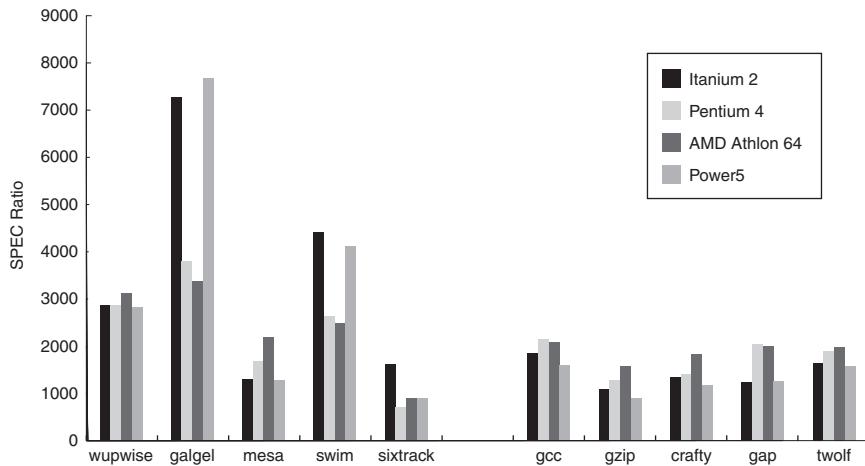
The Itanium 2 processor uses an eight-stage pipeline divided into four major parts:

- *Front-end (stages IPG and Rotate)*—Prefetches up to 32 bytes per clock (two bundles) into a prefetch buffer, which can hold up to eight bundles (24 instructions). Branch prediction is done using a multilevel adaptive predictor like those described in Chapter 3.
- *Instruction delivery (stages EXP and REN)*—Distributes up to six instructions to the 11 functional units. Implements register renaming for both rotation and register stacking.
- *Operand delivery (REG)*—Accesses the register file, performs register bypassing, accesses and updates a register scoreboard, and checks predicate dependences. The scoreboard is used to detect when individual instructions can proceed, so that a stall of one instruction (for example, due to an unpredictable event like a cache miss) in a bundle need not cause the entire bundle to stall. (As we saw in Figure H.8, stalling the entire bundle leads to poor performance unless the instructions are carefully scheduled.)
- *Execution (EXE, DET, and WRB)*—Executes instructions through ALUs and load-store units, detects exceptions and posts NaTs, retires instructions, and performs write-back.

Both the Itanium 1 and the Itanium 2 have many of the features more commonly associated with the dynamically scheduled pipelines described in Chapter 3: dynamic branch prediction, register renaming, scoreboard, a pipeline with a number of stages before execution (to handle instruction alignment, renaming, etc.), and several stages following execution to handle exception detection. Although these mechanisms are generally simpler than those in an advanced dynamically scheduled superscalar, the overall effect is that the Itanium processors, which rely much more on compiler technology, seem to be as complex as the dynamically scheduled processors we saw in Chapter 3!

One might ask why such features are included in a processor that relies primarily on compile time techniques for finding and exploiting parallelism. There are two main motivations. First, dynamic techniques are sometimes significantly better, and omitting them would hurt performance significantly. The inclusion of dynamic branch prediction is such a case.

Second, caches are absolutely necessary to achieve high performance, and with caches come cache misses, which are both unpredictable and which in current processors take a relatively long time. In the early VLIW processors, the entire processor would freeze when a cache miss occurred, retaining the lockstep parallelism initially specified by the compiler. Such an approach is totally unrealistic in a modern processor where cache misses can cost tens to hundreds of cycles. Allowing some instructions to continue while others are stalled, however, requires the introduction of some form of dynamic scheduling, in this case scoreboard. In addition, if a stall is likely to be long, then antidependences are likely to prevent much



**Figure H.11** The performance of four multiple-issue processors for five SPECfp and SPECint benchmarks. The clock rates of the four processors are Itanium 2 at 1.5 GHz, Pentium 4 Extreme Edition at 3.8 GHz, AMD Athlon 64 at 2.8 GHz, and the IBM Power5 at 1.9 GHz.

progress while waiting for the cache miss; hence, the Itanium implementations also introduce register renaming.

### *Itanium 2 Performance*

Figure H.11 shows the performance of a 1.5 GHz Itanium 2 versus a Pentium 4, an AMD Athlon processor, and an IBM Power5 for five SPECint and five SPECfp benchmarks. Overall, the Itanium 2 is slightly slower than the Power5 for the full set of SPEC floating-point benchmarks and about 35% faster than the AMD Athlon or Pentium 4. On SPECint, the Itanium 2 is 15% faster than the Power5, while both the AMD Athlon and Pentium 4 are about 15% faster than the Itanium 2. The Itanium 2 and Power5 are much higher power and have larger die sizes. In fact, the Power5 contains two processors, only one of which is active during normal SPEC benchmarks, and still it has less than half the transistor count of the Itanium. If we were to reduce the die size, transistor count, and power of the Power5 by eliminating one of the processors, the Itanium would be by far the largest and highest-power processor.

## H.7

### Concluding Remarks

When the design of the IA-64 architecture began, it was a joint effort of Hewlett-Packard and Intel and many of the designers had benefited from experience with early VLIW processors as well of years of research building on the early concepts. The clear goal for the IA-64 architecture was to achieve levels of ILP as good or

better than what had been achieved with hardware-based approaches, while also allowing a much simpler hardware implementation. With a simpler hardware implementation, designers hoped that much higher clock rates could be achieved. Indeed, when the IA-64 architecture and the first Itanium were announced, they were announced as the successor to the RISC approaches with clearly superior advantages.

Unfortunately, the practical reality has been quite different. The IA-64 and Itanium implementations appear to be at least as complicated as the dynamically based speculative processors, and neither approach has a significant and consistent performance advantage. The fact that the Itanium designs have also not been more power efficient has led to a situation where the Itanium design has been adopted by only a small number of customers primarily interested in FP performance.

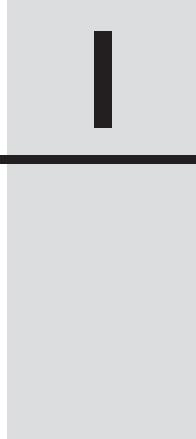
Intel had planned for IA-64 to be its new 64-bit architecture as well. But the combination of its mediocre integer performance (especially in Itanium 1) and large die size, together with AMD's introduction of a 64-bit version of the IA-32 architecture, forced Intel to extend the address space of IA-32. The availability of a larger address space IA-32 processor with strong integer performance has further reduced the interest in IA-64 and Itanium. Most recently, Intel has introduced the name IPF to replace IA-64, since the former name made less sense once the older x86 architecture was extended to 64 bits.

## Reference

- Wilson, R.P., Lam, M.S., 1995. Efficient context-sensitive pointer analysis for C programs. In: Proc. ACM SIGPLAN'95 Conf. on Programming Language Design and Implementation, June 18–21, 1995, La Jolla, Calif, pp. 1–12.

---

|     |                                                                             |      |
|-----|-----------------------------------------------------------------------------|------|
| I.1 | Introduction                                                                | I-2  |
| I.2 | Interprocessor Communication: The Critical Performance Issue                | I-3  |
| I.3 | Characteristics of Scientific Applications                                  | I-6  |
| I.4 | Synchronization: Scaling Up                                                 | I-12 |
| I.5 | Performance of Scientific Applications on Shared-Memory Multiprocessors     | I-21 |
| I.6 | Performance Measurement of Parallel Processors with Scientific Applications | I-33 |
| I.7 | Implementing Cache Coherence                                                | I-34 |
| I.8 | The Custom Cluster Approach: Blue Gene/L                                    | I-41 |
| I.9 | Concluding Remarks                                                          | I-44 |



# Large-Scale Multiprocessors and Scientific Applications

Hennessy and Patterson should move MPPs to Chapter 11.

**Jim Gray, Microsoft Research**  
*when asked about the coverage of massively parallel processors  
(MPPs) for the third edition in 2000*

*Unfortunately for companies in the MPP business, the third edition  
had only ten chapters and the MPP business did not grow as  
anticipated when the first and second edition were written.*

**I.1****Introduction**

The primary application of large-scale multiprocessors is for true parallel programming, as opposed to multiprogramming or transaction-oriented computing where independent tasks are executed in parallel without much interaction. In true parallel computing, a set of tasks execute in a collaborative fashion on one application. The primary target of parallel computing is scientific and technical applications. In contrast, for loosely coupled commercial applications, such as Web servers and most transaction-processing applications, there is little communication among tasks. For such applications, loosely coupled clusters are generally adequate and most cost-effective, since intertask communication is rare.

Because true parallel computing involves cooperating tasks, the nature of communication between those tasks and how such communication is supported in the hardware is of vital importance in determining the performance of the application. The next section of this appendix examines such issues and the characteristics of different communication models.

In comparison to sequential programs, whose performance is largely dictated by the cache behavior and issues related to instruction-level parallelism, parallel programs have several additional characteristics that are important to performance, including the amount of parallelism, the size of parallel tasks, the frequency and nature of intertask communication, and the frequency and nature of synchronization. These aspects are affected both by the underlying nature of the application as well as by the programming style. Section I.3 reviews the important characteristics of several scientific applications to give a flavor of these issues.

As we saw in Chapter 5, synchronization can be quite important in achieving good performance. The larger number of parallel tasks that may need to synchronize makes contention involving synchronization a much more serious problem in large-scale multiprocessors. Section I.4 examines methods of scaling up the synchronization mechanisms of Chapter 5.

Section I.5 explores the detailed performance of shared-memory parallel applications executing on a moderate-scale shared-memory multiprocessor. As we will see, the behavior and performance characteristics are quite a bit more complicated than those in small-scale shared-memory multiprocessors. Section I.6 discusses the general issue of how to examine parallel performance for different sized multiprocessors. Section I.7 explores the implementation challenges of distributed shared-memory cache coherence, the key architectural approach used in moderate-scale multiprocessors. Sections I.7 and I.8 rely on a basic understanding of interconnection networks, and the reader should at least quickly review Appendix F before reading these sections.

Section I.8 explores the design of one of the newest and most exciting large-scale multiprocessors in recent times, Blue Gene. Blue Gene is a cluster-based multiprocessor, but it uses a custom, highly dense node designed specifically for this function, as opposed to the nodes of most earlier cluster multiprocessors that used a node architecture similar to those in a desktop or smaller-scale multiprocessor

node. By using a custom node design, Blue Gene achieves a significant reduction in the cost, physical size, and power consumption of a node. Blue Gene/L, a 64 K-node version, was the world's fastest computer in 2006, as measured by the linear algebra benchmark, Linpack.

## I.2

### Interprocessor Communication: The Critical Performance Issue

In multiprocessors with larger processor counts, interprocessor communication becomes more expensive, since the distance between processors increases. Furthermore, in truly parallel applications where the threads of the application must communicate, there is usually more communication than in a loosely coupled set of distinct processes or independent transactions, which characterize many commercial server applications. These factors combine to make efficient interprocessor communication one of the most important determinants of parallel performance, especially for the scientific market.

Unfortunately, characterizing the communication needs of an application and the capabilities of an architecture is complex. This section examines the key hardware characteristics that determine communication performance, while the next section looks at application behavior and communication needs.

Three performance metrics are critical in any hardware communication mechanism:

1. *Communication bandwidth*—Ideally, the communication bandwidth is limited by processor, memory, and interconnection bandwidths, rather than by some aspect of the communication mechanism. The interconnection network determines the maximum communication capacity of the system. The bandwidth in or out of a single node, which is often as important as total system bandwidth, is affected both by the architecture within the node and by the communication mechanism. How does the communication mechanism affect the communication bandwidth of a node? When communication occurs, resources within the nodes involved in the communication are tied up or occupied, preventing other outgoing or incoming communication. When this *occupancy* is incurred for each word of a message, it sets an absolute limit on the communication bandwidth. This limit is often lower than what the network or memory system can provide. Occupancy may also have a component that is incurred for each communication event, such as an incoming or outgoing request. In the latter case, the occupancy limits the communication rate, and the impact of the occupancy on overall communication bandwidth depends on the size of the messages.
2. *Communication latency*—Ideally, the latency is as low as possible. As Appendix F explains:

$$\begin{aligned}\text{Communication latency} = & \text{ Sender overhead} + \text{Time of flight} \\ & + \text{Transmission time} + \text{Receiver overhead}\end{aligned}$$

assuming no contention. Time of flight is fixed and transmission time is determined by the interconnection network. The software and hardware overheads in sending and receiving messages are largely determined by the communication mechanism and its implementation. Why is latency crucial? Latency affects both performance and how easy it is to program a multiprocessor. Unless latency is hidden, it directly affects performance either by tying up processor resources or by causing the processor to wait.

Overhead and occupancy are closely related, since many forms of overhead also tie up some part of the node, incurring an occupancy cost, which in turn limits bandwidth. Key features of a communication mechanism may directly affect overhead and occupancy. For example, how is the destination address for a remote communication named, and how is protection implemented? When naming and protection mechanisms are provided by the processor, as in a shared address space, the additional overhead is small. Alternatively, if these mechanisms must be provided by the operating system for each communication, this increases the overhead and occupancy costs of communication, which in turn reduce bandwidth and increase latency.

3. *Communication latency hiding*—How well can the communication mechanism hide latency by overlapping communication with computation or with other communication? Although measuring this is not as simple as measuring the first two metrics, it is an important characteristic that can be quantified by measuring the running time on multiprocessors with the same communication latency but different support for latency hiding. Although hiding latency is certainly a good idea, it poses an additional burden on the software system and ultimately on the programmer. Furthermore, the amount of latency that can be hidden is application dependent. Thus, it is usually best to reduce latency wherever possible.

Each of these performance measures is affected by the characteristics of the communications needed in the application, as we will see in the next section. The size of the data items being communicated is the most obvious characteristic, since it affects both latency and bandwidth directly, as well as affecting the efficacy of different latency-hiding approaches. Similarly, the regularity in the communication patterns affects the cost of naming and protection, and hence the communication overhead. In general, mechanisms that perform well with smaller as well as larger data communication requests, and irregular as well as regular communication patterns, are more flexible and efficient for a wider class of applications. Of course, in considering any communication mechanism, designers must consider cost as well as performance.

#### *Advantages of Different Communication Mechanisms*

The two primary means of communicating data in a large-scale multiprocessor are message passing and shared memory. Each of these two primary communication

mechanisms has its advantages. For shared-memory communication, the advantages include

- Compatibility with the well-understood mechanisms in use in centralized multiprocessors, which all use shared-memory communication. The OpenMP consortium (see [www.openmp.org](http://www.openmp.org) for description) has proposed a standardized programming interface for shared-memory multiprocessors. Although message passing also uses a standard, MPI or Message Passing Interface, this standard is not used either in shared-memory multiprocessors or in loosely coupled clusters in use in throughput-oriented environments.
- Ease of programming when the communication patterns among processors are complex or vary dynamically during execution. Similar advantages simplify compiler design.
- The ability to develop applications using the familiar shared-memory model, focusing attention only on those accesses that are performance critical.
- Lower overhead for communication and better use of bandwidth when communicating small items. This arises from the implicit nature of communication and the use of memory mapping to implement protection in hardware, rather than through the I/O system.
- The ability to use hardware-controlled caching to reduce the frequency of remote communication by supporting automatic caching of all data, both shared and private. As we will see, caching reduces both latency and contention for accessing shared data. This advantage also comes with a disadvantage, which we mention below.

The major advantages for message-passing communication include the following:

- The hardware can be simpler, especially by comparison with a scalable shared-memory implementation that supports coherent caching of remote data.
- Communication is explicit, which means it is simpler to understand. In shared-memory models, it can be difficult to know when communication is occurring and when it is not, as well as how costly the communication is.
- Explicit communication focuses programmer attention on this costly aspect of parallel computation, sometimes leading to improved structure in a multi-processor program.
- Synchronization is naturally associated with sending messages, reducing the possibility for errors introduced by incorrect synchronization.
- It makes it easier to use sender-initiated communication, which may have some advantages in performance.
- If the communication is less frequent and more structured, it is easier to improve fault tolerance by using a transaction-like structure. Furthermore,

the less tight coupling of nodes and explicit communication make fault isolation simpler.

- The very largest multiprocessors use a cluster structure, which is inherently based on message passing. Using two different communication models may introduce more complexity than is warranted.

Of course, the desired communication model can be created in software on top of a hardware model that supports either of these mechanisms. Supporting message passing on top of shared memory is considerably easier: Because messages essentially send data from one memory to another, sending a message can be implemented by doing a copy from one portion of the address space to another. The major difficulties arise from dealing with messages that may be misaligned and of arbitrary length in a memory system that is normally oriented toward transferring aligned blocks of data organized as cache blocks. These difficulties can be overcome either with small performance penalties in software or with essentially no penalties, using a small amount of hardware support.

Supporting shared memory efficiently on top of hardware for message passing is much more difficult. Without explicit hardware support for shared memory, all shared-memory references need to involve the operating system to provide address translation and memory protection, as well as to translate memory references into message sends and receives. Loads and stores usually move small amounts of data, so the high overhead of handling these communications in software severely limits the range of applications for which the performance of software-based shared memory is acceptable. For these reasons, it has never been practical to use message passing to implement shared memory for a commercial system.

### I.3

## Characteristics of Scientific Applications

The primary use of scalable shared-memory multiprocessors is for true parallel programming, as opposed to multiprogramming or transaction-oriented computing. The primary target of parallel computing is scientific and technical applications. Thus, understanding the design issues requires some insight into the behavior of such applications. This section provides such an introduction.

### Characteristics of Scientific Applications

Our scientific/technical parallel workload consists of two applications and two computational kernels. The kernels are fast Fourier transformation (FFT) and an LU decomposition, which were chosen because they represent commonly used techniques in a wide variety of applications and have performance characteristics typical of many parallel scientific applications. In addition, the kernels have small code segments whose behavior we can understand and directly track to specific architectural characteristics. Like many scientific applications, I/O is essentially nonexistent in this workload.

The two applications that we use in this appendix are Barnes and Ocean, which represent two important but very different types of parallel computation. We briefly describe each of these applications and kernels and characterize their basic behavior in terms of parallelism and communication. We describe how the problem is decomposed for a distributed shared-memory multiprocessor; certain data decompositions that we describe are not necessary on multiprocessors that have a single, centralized memory.

### *The FFT Kernel*

The FFT is the key kernel in applications that use spectral methods, which arise in fields ranging from signal processing to fluid flow to climate modeling. The FFT application we study here is a one-dimensional version of a parallel algorithm for a complex number FFT. It has a sequential execution time for  $n$  data points of  $n \log n$ . The algorithm uses a high radix (equal to  $\sqrt{n}$ ) that minimizes communication. The measurements shown in this appendix are collected for a million-point input data set.

There are three primary data structures: the input and output arrays of the data being transformed and the roots of unity matrix, which is precomputed and only read during the execution. All arrays are organized as square matrices. The six steps in the algorithm are as follows:

1. Transpose data matrix.
2. Perform 1D FFT on each row of data matrix.
3. Multiply the roots of unity matrix by the data matrix and write the result in the data matrix.
4. Transpose data matrix.
5. Perform 1D FFT on each row of data matrix.
6. Transpose data matrix.

The data matrices and the roots of unity matrix are partitioned among processors in contiguous chunks of rows, so that each processor's partition falls in its own local memory. The first row of the roots of unity matrix is accessed heavily by all processors and is often replicated, as we do, during the first step of the algorithm just shown. The data transposes ensure good locality during the individual FFT steps, which would otherwise access nonlocal data.

The only communication is in the transpose phases, which require all-to-all communication of large amounts of data. Contiguous subcolumns in the rows assigned to a processor are grouped into blocks, which are transposed and placed into the proper location of the destination matrix. Every processor transposes one block locally and sends one block to each of the other processors in the system. Although there is no reuse of individual words in the transpose, with long cache blocks it makes sense to block the transpose to take advantage of the spatial locality afforded by long blocks in the source matrix.

### The LU Kernel

LU is an LU factorization of a dense matrix and is representative of many dense linear algebra computations, such as QR factorization, Cholesky factorization, and eigenvalue methods. For a matrix of size  $n \times n$  the running time is  $n^3$  and the parallelism is proportional to  $n^2$ . Dense LU factorization can be performed efficiently by blocking the algorithm, using the techniques in Chapter 2, which leads to highly efficient cache behavior and low communication. After blocking the algorithm, the dominant computation is a dense matrix multiply that occurs in the innermost loop. The block size is chosen to be small enough to keep the cache miss rate low and large enough to reduce the time spent in the less parallel parts of the computation. Relatively small block sizes ( $8 \times 8$  or  $16 \times 16$ ) tend to satisfy both criteria.

Two details are important for reducing interprocessor communication. First, the blocks of the matrix are assigned to processors using a 2D tiling: The  $\frac{n}{B} \times \frac{n}{B}$  (where each block is  $B \times B$ ) matrix of blocks is allocated by laying a grid of size  $p \times p$  over the matrix of blocks in a cookie-cutter fashion until all the blocks are allocated to a processor. Second, the dense matrix multiplication is performed by the processor that owns the *destination* block. With this blocking and allocation scheme, communication during the reduction is both regular and predictable. For the measurements in this appendix, the input is a  $512 \times 512$  matrix and a block of  $16 \times 16$  is used.

A natural way to code the blocked LU factorization of a 2D matrix in a shared address space is to use a 2D array to represent the matrix. Because blocks are allocated in a tiled decomposition, and a block is not contiguous in the address space in a 2D array, it is very difficult to allocate blocks in the local memories of the processors that own them. The solution is to ensure that blocks assigned to a processor are allocated locally and contiguously by using a 4D array (with the first two dimensions specifying the block number in the 2D grid of blocks, and the next two specifying the element in the block).

### The Barnes Application

Barnes is an implementation of the Barnes-Hut  $n$ -body algorithm solving a problem in galaxy evolution. *N-body algorithms* simulate the interaction among a large number of bodies that have forces interacting among them. In this instance, the bodies represent collections of stars and the force is gravity. To reduce the computational time required to model completely all the individual interactions among the bodies, which grow as  $n^2$ ,  $n$ -body algorithms take advantage of the fact that the forces drop off with distance. (Gravity, for example, drops off as  $1/d^2$ , where  $d$  is the distance between the two bodies.) The Barnes-Hut algorithm takes advantage of this property by treating a collection of bodies that are “far away” from another body as a single point at the center of mass of the collection and with mass equal to the collection. If the body is far enough from any body in the collection, then the error introduced will be

negligible. The collections are structured in a hierarchical fashion, which can be represented in a tree. This algorithm yields an  $n \log n$  running time with parallelism proportional to  $n$ .

The Barnes-Hut algorithm uses an octree (each node has up to eight children) to represent the eight cubes in a portion of space. Each node then represents the collection of bodies in the subtree rooted at that node, which we call a *cell*. Because the density of space varies and the leaves represent individual bodies, the depth of the tree varies. The tree is traversed once per body to compute the net force acting on that body. The force calculation algorithm for a body starts at the root of the tree. For every node in the tree it visits, the algorithm determines if the center of mass of the cell represented by the subtree rooted at the node is “far enough away” from the body. If so, the entire subtree under that node is approximated by a single point at the center of mass of the cell, and the force that this center of mass exerts on the body is computed. On the other hand, if the center of mass is not far enough away, the cell must be “opened” and each of its subtrees visited. The distance between the body and the cell, together with the error tolerances, determines which cells must be opened. This force calculation phase dominates the execution time. This appendix takes measurements using 16K bodies; the criterion for determining whether a cell needs to be opened is set to the middle of the range typically used in practice.

Obtaining effective parallel performance on Barnes-Hut is challenging because the distribution of bodies is nonuniform and changes over time, making partitioning the work among the processors and maintenance of good locality of reference difficult. We are helped by two properties: (1) the system evolves slowly, and (2) because gravitational forces fall off quickly, with high probability, each cell requires touching a small number of other cells, most of which were used on the last time step. The tree can be partitioned by allocating each processor a subtree. Many of the accesses needed to compute the force on a body in the subtree will be to other bodies in the subtree. Since the amount of work associated with a subtree varies (cells in dense portions of space will need to access more cells), the size of the subtree allocated to a processor is based on some measure of the work it has to do (e.g., how many other cells it needs to visit), rather than just on the number of nodes in the subtree. By partitioning the octree representation, we can obtain good load balance and good locality of reference, while keeping the partitioning cost low. Although this partitioning scheme results in good locality of reference, the resulting data references tend to be for small amounts of data and are unstructured. Thus, this scheme requires an efficient implementation of shared-memory communication.

### *The Ocean Application*

Ocean simulates the influence of eddy and boundary currents on large-scale flow in the ocean. It uses a restricted red-black Gauss-Seidel multigrid technique to solve a set of elliptical partial differential equations. *Red-black Gauss-Seidel* is an iteration technique that colors the points in the grid so as to consistently update

each point based on previous values of the adjacent neighbors. *Multigrid methods* solve finite difference equations by iteration using hierarchical grids. Each grid in the hierarchy has fewer points than the grid below and is an approximation to the lower grid. A finer grid increases accuracy and thus the rate of convergence, while requiring more execution time, since it has more data points. Whether to move up or down in the hierarchy of grids used for the next iteration is determined by the rate of change of the data values. The estimate of the error at every time step is used to decide whether to stay at the same grid, move to a coarser grid, or move to a finer grid. When the iteration converges at the finest level, a solution has been reached. Each iteration has  $n^2$  work for an  $n \times n$  grid and the same amount of parallelism.

The arrays representing each grid are dynamically allocated and sized to the particular problem. The entire ocean basin is partitioned into square subgrids (as close as possible) that are allocated in the portion of the address space corresponding to the local memory of the individual processors, which are assigned responsibility for the subgrid. For the measurements in this appendix we use an input that has  $130 \times 130$  grid points. There are five steps in a time iteration. Since data are exchanged between the steps, all the processors present synchronize at the end of each step before proceeding to the next. Communication occurs when the boundary points of a subgrid are accessed by the adjacent subgrid in nearest-neighbor fashion.

### *Computation/Communication for the Parallel Programs*

A key characteristic in determining the performance of parallel programs is the ratio of computation to communication. If the ratio is high, it means the application has lots of computation for each datum communicated. As we saw in Section I.2, communication is the costly part of parallel computing; therefore, high computation-to-communication ratios are very beneficial. In a parallel processing environment, we are concerned with how the ratio of computation to communication changes as we increase either the number of processors, the size of the problem, or both. Knowing how the ratio changes as we increase the processor count sheds light on how well the application can be sped up. Because we are often interested in running larger problems, it is vital to understand how changing the data set size affects this ratio.

To understand what happens quantitatively to the computation-to-communication ratio as we add processors, consider what happens separately to computation and to communication as we either add processors or increase problem size. Figure I.1 shows that as we add processors, for these applications, the amount of computation per processor falls proportionately and the amount of communication per processor falls more slowly. As we increase the problem size, the computation scales as the  $O()$  complexity of the algorithm dictates. Communication scaling is more complex and depends on details of the algorithm; we describe the basic phenomena for each application in the caption of Figure I.1.

| Application | Scaling of computation | Scaling of communication                          | Scaling of computation-to-communication   |
|-------------|------------------------|---------------------------------------------------|-------------------------------------------|
| FFT         | $\frac{n \log n}{p}$   | $\frac{n}{p}$                                     | $\log n$                                  |
| LU          | $\frac{n}{p}$          | $\frac{\sqrt{n}}{\sqrt{p}}$                       | $\frac{\sqrt{n}}{\sqrt{p}}$               |
| Barnes      | $\frac{n \log n}{p}$   | approximately $\frac{\sqrt{n}(\log n)}{\sqrt{p}}$ | approximately $\frac{\sqrt{n}}{\sqrt{p}}$ |
| Ocean       | $\frac{n}{p}$          | $\frac{\sqrt{n}}{\sqrt{p}}$                       | $\frac{\sqrt{n}}{\sqrt{p}}$               |

**Figure I.1 Scaling of computation, of communication, and of the ratio are critical factors in determining performance on parallel multiprocessors.** In this table,  $p$  is the increased processor count and  $n$  is the increased dataset size. Scaling is on a per-processor basis. The computation scales up with  $n$  at the rate given by  $O()$  analysis and scales down linearly as  $p$  is increased. Communication scaling is more complex. In FFT, all data points must interact, so communication increases with  $n$  and decreases with  $p$ . In LU and Ocean, communication is proportional to the boundary of a block, so it scales with dataset size at a rate proportional to the side of a square with  $n$  points, namely,  $\sqrt{n}$ ; for the same reason communication in these two applications scales inversely to  $\sqrt{p}$ . Barnes has the most complex scaling properties. Because of the fall-off of interaction between bodies, the basic number of interactions among bodies that require communication scales as  $\sqrt{n}$ . An additional factor of  $\log n$  is needed to maintain the relationships among the bodies. As processor count is increased, communication scales inversely to  $\sqrt{p}$ .

The overall computation-to-communication ratio is computed from the individual growth rate in computation and communication. In general, this ratio rises slowly with an increase in dataset size and decreases as we add processors. This reminds us that performing a fixed-size problem with more processors leads to increasing inefficiencies because the amount of communication among processors grows. It also tells us how quickly we must scale dataset size as we add processors to keep the fraction of time in communication fixed. The following example illustrates these trade-offs.

**Example** Suppose we know that for a given multiprocessor the Ocean application spends 20% of its execution time waiting for communication when run on four processors. Assume that the cost of each communication event is independent of processor count, which is not true in general, since communication costs rise with processor count. How much faster might we expect Ocean to run on a 32-processor machine with the same problem size? What fraction of the execution time is spent on communication in this case? How much larger a problem should we run if we want the fraction of time spent communicating to be the same?

**Answer** The computation-to-communication ratio for Ocean is  $\sqrt{n}/\sqrt{p}$ , so if the problem size is the same, the communication frequency scales by  $\sqrt{p}$ . This means that communication time increases by  $\sqrt{8}$ . We can use a variation on Amdahl's law,

recognizing that the computation is decreased but the communication time is increased. If  $T_4$  is the total execution time for four processors, then the execution time for 32 processors is

$$\begin{aligned} T_{32} &= \text{Compute time} + \text{Communication time} \\ &= \frac{0.8 \times T_4}{8} + (0.2 \times T_4) \times \sqrt{8} \\ &= 0.1 \times T_4 + 0.57 \times T_4 = 0.67 \times T_4 \end{aligned}$$

Hence, the speedup is

$$\text{Speedup} = \frac{T_4}{T_{32}} = \frac{T_4}{0.67 \times T_4} = 1.49$$

and the fraction of time spent in communication goes from 20% to  $0.57/0.67 = 85\%$ .

For the fraction of the communication time to remain the same, we must keep the computation-to-communication ratio the same, so the problem size must scale at the same rate as the processor count. Notice that, because we have changed the problem size, we cannot fairly compare the speedup of the original problem and the scaled problem. We will return to the critical issue of scaling applications for multiprocessors in Section I.6.

---

## I.4

## Synchronization: Scaling Up

In this section, we focus first on synchronization performance problems in larger multiprocessors and then on solutions for those problems.

### Synchronization Performance Challenges

To understand why the simple spin lock scheme presented in Chapter 5 does not scale well, imagine a large multiprocessor with all processors contending for the same lock. The directory or bus acts as a point of serialization for all the processors, leading to lots of contention, as well as traffic. The following example shows how bad things can be.

---

**Example** Suppose there are 10 processors on a bus and each tries to lock a variable simultaneously. Assume that each bus transaction (read miss or write miss) is 100 clock cycles long. You can ignore the time of the actual read or write of a lock held in the cache, as well as the time the lock is held (they won't matter much!). Determine the number of bus transactions required for all 10 processors to acquire the lock, assuming they are all spinning when the lock is released at time 0. About how long will it take to process the 10 requests? Assume that the bus is

totally fair so that every pending request is serviced before a new request and that the processors are equally fast.

**Answer** When  $i$  processes are contending for the lock, they perform the following sequence of actions, each of which generates a bus transaction:

- $i$  load linked operations to access the lock
- $i$  store conditional operations to try to lock the lock
- 1 store (to release the lock)

Thus, for  $i$  processes, there are a total of  $2i+1$  bus transactions. Note that this assumes that the critical section time is negligible, so that the lock is released before any other processors whose store conditional failed attempt another load linked.

Thus, for  $n$  processes, the total number of bus operations is

$$\sum_{i=1}^n (2i+1) = n(n+1) + n = n^2 + 2n$$

For 10 processes there are 120 bus transactions requiring 12,000 clock cycles or 120 clock cycles per lock acquisition!

---

The difficulty in this example arises from contention for the lock and serialization of lock access, as well as the latency of the bus access. (The fairness property of the bus actually makes things worse, since it delays the processor that claims the lock from releasing it; unfortunately, for any bus arbitration scheme some worst-case scenario does exist.) The key advantages of spin locks—that they have low overhead in terms of bus or network cycles and offer good performance when locks are reused by the same processor—are both lost in this example. We will consider alternative implementations in the next section, but before we do that, let's consider the use of spin locks to implement another common high-level synchronization primitive.

### Barrier Synchronization

One additional common synchronization operation in programs with parallel loops is a *barrier*. A barrier forces all processes to wait until all the processes reach the barrier and then releases all of the processes. A typical implementation of a barrier can be done with two spin locks: one to protect a counter that tallies the processes arriving at the barrier and one to hold the processes until the last process arrives at the barrier. To implement a barrier, we usually use the ability to spin on a variable until it satisfies a test; we use the notation `spin(condition)` to indicate this. Figure I.2 is a typical implementation, assuming that `lock` and `unlock` provide basic spin locks and `total` is the number of processes that must reach the barrier.

```

lock(counterlock);/* ensure update atomic */
if(count==0) release=0; /* first=>reset release */
count = count + 1; /* count arrivals */
unlock(counterlock);/* release lock */
if(count==total) {/* all arrived */
 count=0; /* reset counter */
 release=1; /* release processes */
}
else /* more to come */
 spin(release==1); /* wait for arrivals */
}

```

---

**Figure I.2** Code for a simple barrier. The lock `counterlock` protects the counter so that it can be atomically incremented. The variable `count` keeps the tally of how many processes have reached the barrier. The variable `release` is used to hold the processes until the last one reaches the barrier. The operation `spin (release==1)` causes a process to wait until all processes reach the barrier.

In practice, another complication makes barrier implementation slightly more complex. Frequently a barrier is used within a loop, so that processes released from the barrier would do some work and then reach the barrier again. Assume that one of the processes never actually leaves the barrier (it stays at the spin operation), which could happen if the OS scheduled another process, for example. Now it is possible that one process races ahead and gets to the barrier again before the last process has left. The “fast” process then traps the remaining “slow” process in the barrier by resetting the flag `release`. Now all the processes will wait infinitely at the next instance of this barrier because one process is trapped at the last instance, and the number of processes can never reach the value of `total`.

The important observation in this example is that the programmer did nothing wrong. Instead, the implementer of the barrier made some assumptions about forward progress that cannot be assumed. One obvious solution to this is to count the processes as they exit the barrier (just as we did on entry) and not to allow any process to reenter and reinitialize the barrier until all processes have left the prior instance of this barrier. This extra step would significantly increase the latency of the barrier and the contention, which as we will see shortly are already large. An alternative solution is a *sense-reversing barrier*, which makes use of a private per-process variable, `local_sense`, which is initialized to 1 for each process. Figure I.3 shows the code for the sense-reversing barrier. This version of a barrier is safely usable; as the next example shows, however, its performance can still be quite poor.

```

local_sense =! local_sense; /* toggle local_sense */
lock (counterlock);/* ensure update atomic */
count=count+1; /* count arrivals */
if (count==total) {/* all arrived */
 count=0; /* reset counter */
 release=local_sense; /* release processes */
}
unlock (counterlock);/* unlock */
spin (release==local_sense); /* wait for signal */
}

```

**Figure I.3** Code for a sense-reversing barrier. The key to making the barrier reusable is the use of an alternating pattern of values for the flag `release`, which controls the exit from the barrier. If a process races ahead to the next instance of this barrier while some other processes are still in the barrier, the fast process cannot trap the other processes, since it does not reset the value of `release` as it did in Figure I.2.

---

**Example** Suppose there are 10 processors on a bus and each tries to execute a barrier simultaneously. Assume that each bus transaction is 100 clock cycles, as before. You can ignore the time of the actual read or write of a lock held in the cache as the time to execute other nonsynchronization operations in the barrier implementation. Determine the number of bus transactions required for all 10 processors to reach the barrier, be released from the barrier, and exit the barrier. Assume that the bus is totally fair, so that every pending request is serviced before a new request and that the processors are equally fast. Don't worry about counting the processors out of the barrier. How long will the entire process take?

**Answer** We assume that load linked and store conditional are used to implement lock and unlock. Figure I.4 shows the sequence of bus events for a processor to traverse the barrier, assuming that the first process to grab the bus does not have the lock. There is a slight difference for the last process to reach the barrier, as described in the caption.

For the  $i$ th process, the number of bus transactions is  $3i + 4$ . The last process to reach the barrier requires one less. Thus, for  $n$  processes, the number of bus transactions is

$$\left( \sum_{i=1}^n (3i + 4) \right) - 1 = \frac{3n^2 + 11n}{2} - 1$$

For 10 processes, this is 204 bus cycles or 20,400 clock cycles! Our barrier operation takes almost twice as long as the 10-processor lock-unlock sequence.

---

| Event             | Number of times for process $i$ | Corresponding source line                   | Comment                                                  |
|-------------------|---------------------------------|---------------------------------------------|----------------------------------------------------------|
| LL counterlock    | $i$                             | <code>lock(counterlock);</code>             | All processes try for lock.                              |
| Store conditional | $i$                             | <code>lock(counterlock);</code>             | All processes try for lock.                              |
| LD count          | 1                               | <code>count = count + 1;</code>             | Successful process.                                      |
| Load linked       | $i - 1$                         | <code>lock(counterlock);</code>             | Unsuccessful process; try again.                         |
| SD count          | 1                               | <code>count = count + 1;</code>             | Miss to get exclusive access.                            |
| SD counterlock    | 1                               | <code>unlock(counterlock);</code>           | Miss to get the lock.                                    |
| LD release        | 2                               | <code>spin (release==local_sense); /</code> | Read release: misses initially and when finally written. |

**Figure I.4** Here are the actions, which require a bus transaction, taken when the  $i$ th process reaches the barrier. The last process to reach the barrier requires one less bus transaction, since its read of release for the spin will hit in the cache!

As we can see from these examples, synchronization performance can be a real bottleneck when there is substantial contention among multiple processes. When there is little contention and synchronization operations are infrequent, we are primarily concerned about the latency of a synchronization primitive—that is, how long it takes an individual process to complete a synchronization operation. Our basic spin lock operation can do this in two bus cycles: one to initially read the lock and one to write it. We could improve this to a single bus cycle by a variety of methods. For example, we could simply spin on the swap operation. If the lock were almost always free, this could be better, but if the lock were not free, it would lead to lots of bus traffic, since each attempt to lock the variable would lead to a bus cycle. In practice, the latency of our spin lock is not quite as bad as we have seen in this example, since the write miss for a data item present in the cache is treated as an upgrade and will be cheaper than a true read miss.

The more serious problem in these examples is the serialization of each process's attempt to complete the synchronization. This serialization is a problem when there is contention because it greatly increases the time to complete the synchronization operation. For example, if the time to complete all 10 lock and unlock operations depended only on the latency in the uncontended case, then it would take 1000 rather than 15,000 cycles to complete the synchronization operations. The barrier situation is as bad, and in some ways worse, since it is highly likely to incur contention. The use of a bus interconnect exacerbates these problems, but serialization could be just as serious in a directory-based multiprocessor, where the latency would be large. The next subsection presents some solutions that are useful when either the contention is high or the processor count is large.

## Synchronization Mechanisms for Larger-Scale Multiprocessors

What we would like are synchronization mechanisms that have low latency in uncontended cases and that minimize serialization in the case where contention is significant. We begin by showing how software implementations can improve the performance of locks and barriers when contention is high; we then explore two basic hardware primitives that reduce serialization while keeping latency low.

### Software Implementations

The major difficulty with our spin lock implementation is the delay due to contention when many processes are spinning on the lock. One solution is to artificially delay processes when they fail to acquire the lock. The best performance is obtained by increasing the delay exponentially whenever the attempt to acquire the lock fails. Figure I.5 shows how a spin lock with *exponential back-off* is implemented. Exponential back-off is a common technique for reducing contention in shared resources, including access to shared networks and buses (see Sections F.4 to F.8). This implementation still attempts to preserve low latency when contention is small by not delaying the initial spin loop. The result is that if many processes are waiting, the back-off does not affect the processes on their first attempt to acquire the lock. We could also delay that process, but the result would be poorer

```

lockit: DADDUI R3,R0,#1 ;R3 = initial delay
 LL R2,0(R1) ;load linked
 BNEZ R2,lockit ;not available-spin
 DADDUI R2,R2,#1 ;get locked value
 SC R2,0(R1) ;store conditional
 BNEZ R2,gotit ;branch if store succeeds
 DSLL R3,R3,#1 ;increased delay by factor of 2
 PAUSE R3 ;delays by value in R3
 J lockit
gotit: use data protected by lock

```

---

**Figure I.5 A spin lock with exponential back-off.** When the store conditional fails, the process delays itself by the value in R3. The delay can be implemented by decrementing a copy of the value in R3 until it reaches 0. The exact timing of the delay is multiprocessor dependent, although it should start with a value that is approximately the time to perform the critical section and release the lock. The statement pause R3 should cause a delay of R3 of these time units. The value in R3 is increased by a factor of 2 every time the store conditional fails, which causes the process to wait twice as long before trying to acquire the lock again. The small variations in the rate at which competing processors execute instructions are usually sufficient to ensure that processes will not continually collide. If the natural perturbation in execution time was insufficient, R3 could be initialized with a small random value, increasing the variance in the successive delays and reducing the probability of successive collisions.

performance when the lock was in use by only two processes and the first one happened to find it locked.

Another technique for implementing locks is to use queuing locks. Queuing locks work by constructing a queue of waiting processors; whenever a processor frees up the lock, it causes the next processor in the queue to attempt access. This eliminates contention for a lock when it is freed. We show how queuing locks operate in the next section using a hardware implementation, but software implementations using arrays can achieve most of the same benefits. Before we look at hardware primitives, let's look at a better mechanism for barriers.

Our barrier implementation suffers from contention both during the *gather* stage, when we must atomically update the count, and at the *release* stage, when all the processes must read the release flag. The former is more serious because it requires exclusive access to the synchronization variable and thus creates much more serialization; in comparison, the latter generates only read contention. We can reduce the contention by using a *combining tree*, a structure where multiple requests are locally combined in tree fashion. The same combining tree can be used to implement the release process, reducing the contention there.

Our combining tree barrier uses a predetermined  $n$ -ary tree structure. We use the variable  $k$  to stand for the fan-in; in practice,  $k=4$  seems to work well. When the  $k$ th process arrives at a node in the tree, we signal the next level in the tree. When a process arrives at the root, we release all waiting processes. As in our earlier example, we use a sense-reversing technique. A tree-based barrier, as shown in Figure I.6, uses a tree to combine the processes and a single signal to release the barrier. Some MPPs (e.g., the T3D and CM-5) have also included hardware support for barriers, but more recent machines have relied on software libraries for this support.

### *Hardware Primitives*

In this subsection, we look at two hardware synchronization primitives. The first primitive deals with locks, while the second is useful for barriers and a number of other user-level operations that require counting or supplying distinct indices. In both cases, we can create a hardware primitive where latency is essentially identical to our earlier version, but with much less serialization, leading to better scaling when there is contention.

The major problem with our original lock implementation is that it introduces a large amount of unneeded contention. For example, when the lock is released all processors generate both a read and a write miss, although at most one processor can successfully get the lock in the unlocked state. This sequence happens on each of the 10 lock/unlock sequences, as we saw in the example on page I-12.

We can improve this situation by explicitly handing the lock from one waiting processor to the next. Rather than simply allowing all processors to compete every time the lock is released, we keep a list of the waiting processors and hand the lock to one explicitly, when its turn comes. This sort of mechanism has been called a *queuing lock*. Queuing locks can be implemented either in hardware, which we

```

struct node /* a node in the combining tree */
 int counterlock; /* lock for this node */
 int count; /* counter for this node */
 int parent; /* parent in the tree=0..P-1 except for root */
};
struct node tree [0..P-1]; /* the tree of nodes */
int local_sense; /* private per processor */
int release; /* global release flag */

/* function to implement barrier */
barrier (int mynode, int local_sense) {
 lock (tree[mynode].counterlock); /* protect count */
 tree[mynode].count=tree[mynode].count+1;
 /* increment count */
 if (tree[mynode].count==k) /* all arrived at mynode */
 if (tree[mynode].parent >=0) {
 barrier(tree[mynode].parent);
 } else
 release = local_sense;
 };
 tree[mynode].count=0; /* reset for the next time */
 unlock (tree[mynode].counterlock); /* unlock */
 spin (release==local_sense); /* wait */
};
/* code executed by a processor to join barrier */
local_sense =! local_sense;
barrier (mynode);

```

---

**Figure I.6 An implementation of a tree-based barrier reduces contention considerably.** The tree is assumed to be prebuilt statically using the nodes in the array tree. Each node in the tree combines  $k$  processes and provides a separate counter and lock, so that at most  $k$  processes contend at each node. When the  $k$ th process reaches a node in the tree, it goes up to the parent, incrementing the count at the parent. When the count in the parent node reaches  $k$ , the release flag is set. The count in each node is reset by the last process to arrive. Sense-reversing is used to avoid races as in the simple barrier. The value of tree[root].parent should be set to  $-1$  when the tree is initially built.

describe here, or in software using an array to keep track of the waiting processes. The basic concepts are the same in either case. Our hardware implementation assumes a directory-based multiprocessor where the individual processor caches are addressable. In a bus-based multiprocessor, a software implementation would be more appropriate and would have each processor using a different address for the lock, permitting the explicit transfer of the lock from one process to another.

How does a queuing lock work? On the first miss to the lock variable, the miss is sent to a synchronization controller, which may be integrated with the memory controller (in a bus-based system) or with the directory controller. If the lock is free, it is simply returned to the processor. If the lock is unavailable,

the controller creates a record of the node's request (such as a bit in a vector) and sends the processor back a locked value for the variable, which the processor then spins on. When the lock is freed, the controller selects a processor to go ahead from the list of waiting processors. It can then either update the lock variable in the selected processor's cache or invalidate the copy, causing the processor to miss and fetch an available copy of the lock.

---

**Example** How many bus transactions and how long does it take to have 10 processors lock and unlock the variable using a queuing lock that updates the lock on a miss? Make the other assumptions about the system the same as those in the earlier example on page I-12.

**Answer** For  $n$  processors, each will initially attempt a lock access, generating a bus transaction; one will succeed and free up the lock, for a total of  $n + 1$  transactions for the first processor. Each subsequent processor requires two bus transactions, one to receive the lock and one to free it up. Thus, the total number of bus transactions is  $(n + 1) + 2(n - 1) = 3n - 1$ . Note that the number of bus transactions is now linear in the number of processors contending for the lock, rather than quadratic, as it was with the spin lock we examined earlier. For 10 processors, this requires 29 bus cycles or 2900 clock cycles.

---

There are a couple of key insights in implementing such a queuing lock capability. First, we need to be able to distinguish the initial access to the lock, so we can perform the queuing operation, and also the lock release, so we can provide the lock to another processor. The queue of waiting processes can be implemented by a variety of mechanisms. In a directory-based multiprocessor, this queue is akin to the sharing set, and similar hardware can be used to implement the directory and queuing lock operations. One complication is that the hardware must be prepared to reclaim such locks, since the process that requested the lock may have been context-switched and may not even be scheduled again on the same processor.

Queuing locks can be used to improve the performance of our barrier operation. Alternatively, we can introduce a primitive that reduces the amount of time needed to increment the barrier count, thus reducing the serialization at this bottleneck, which should yield comparable performance to using queuing locks. One primitive that has been introduced for this and for building other synchronization operations is *fetch-and-increment*, which atomically fetches a variable and increments its value. The returned value can be either the incremented value or the fetched value. Using *fetch-and-increment* we can dramatically improve our barrier implementation, compared to the simple code-sensing barrier.

---

**Example** Write the code for the barrier using *fetch-and-increment*. Making the same assumptions as in our earlier example and also assuming that a *fetch-and-increment* operation, which returns the incremented value, takes 100 clock cycles, determine the time for 10 processors to traverse the barrier. How many bus cycles are required?

```

local_sense =! local_sense; /* toggle local_sense */
fetch_and_increment(count);/* atomic update */
if (count==total) {/* all arrived */
 count=0; /* reset counter */
 release=local_sense; /* release processes */
}
else {/* more to come */
 spin (release==local_sense); /* wait for signal */
}

```

**Figure I.7** Code for a sense-reversing barrier using fetch-and-increment to do the counting.

**Answer** Figure I.7 shows the code for the barrier. For  $n$  processors, this implementation requires  $n$  fetch-and-increment operations,  $n$  cache misses to access the count, and  $n$  cache misses for the release operation, for a total of  $3n$  bus transactions. For 10 processors, this is 30 bus transactions or 3000 clock cycles. Like the queuing lock, the time is linear in the number of processors. Of course, fetch-and-increment can also be used in implementing the combining tree barrier, reducing the serialization at each node in the tree.

As we have seen, synchronization problems can become quite acute in largerscale multiprocessors. When the challenges posed by synchronization are combined with the challenges posed by long memory latency and potential load imbalance in computations, we can see why getting efficient usage of large-scale parallel processors is very challenging.

## I.5

## Performance of Scientific Applications on Shared-Memory Multiprocessors

This section covers the performance of the scientific applications from Section I.3 on both symmetric shared-memory and distributed shared-memory multiprocessors.

### Performance of a Scientific Workload on a Symmetric Shared-Memory Multiprocessor

We evaluate the performance of our four scientific applications on a symmetric shared-memory multiprocessor using the following problem sizes:

- *Barnes-Hut*—16 K bodies run for six time steps (the accuracy control is set to 1.0, a typical, realistic value)

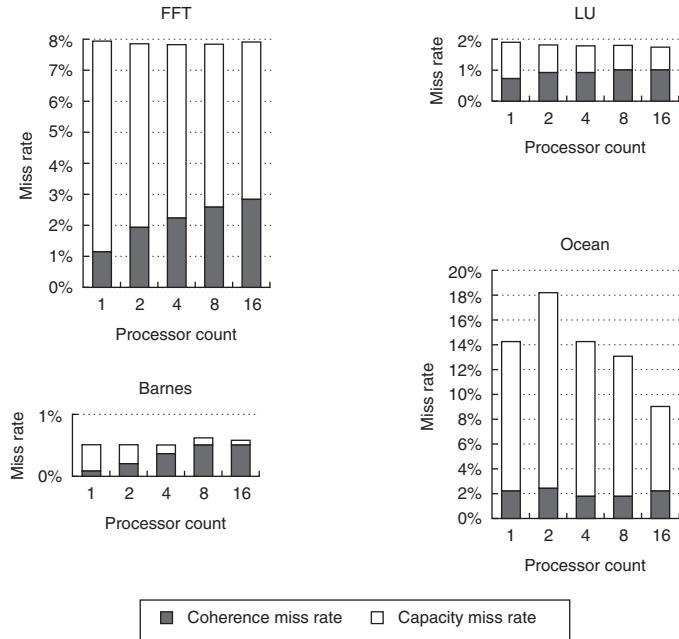
- *FFT*—1 million complex data points
- *LU*—A  $512 \times 512$  matrix is used with  $16 \times 16$  blocks
- *Ocean*—A  $130 \times 130$  grid with a typical error tolerance

In looking at the miss rates as we vary processor count, cache size, and block size, we decompose the total miss rate into *coherence misses* and normal uniprocessor misses. The normal uniprocessor misses consist of capacity, conflict, and compulsory misses. We label these misses as capacity misses because that is the dominant cause for these benchmarks. For these measurements, we include as a coherence miss any write misses needed to upgrade a block from shared to exclusive, even though no one is sharing the cache block. This measurement reflects a protocol that does not distinguish between a private and shared cache block.

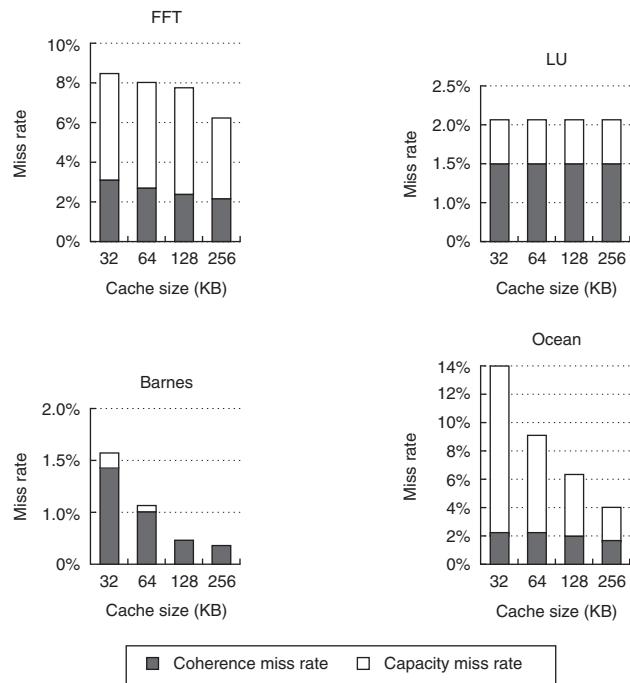
Figure I.8 shows the data miss rates for our four applications, as we increase the number of processors from 1 to 16, while keeping the problem size constant. As we increase the number of processors, the total amount of cache increases, usually causing the capacity misses to drop. In contrast, increasing the processor count usually causes the amount of communication to increase, in turn causing the coherence misses to rise. The magnitude of these two effects differs by application.

In FFT, the capacity miss rate drops (from nearly 7% to just over 5%) but the coherence miss rate increases (from about 1% to about 2.7%), leading to a constant overall miss rate. Ocean shows a combination of effects, including some that relate to the partitioning of the grid and how grid boundaries map to cache blocks. For a typical 2D grid code the communication-generated misses are proportional to the boundary of each partition of the grid, while the capacity misses are proportional to the area of the grid. Therefore, increasing the total amount of cache while keeping the total problem size fixed will have a more significant effect on the capacity miss rate, at least until each subgrid fits within an individual processor's cache. The significant jump in miss rate between one and two processors occurs because of conflicts that arise from the way in which the multiple grids are mapped to the caches. This conflict is present for direct-mapped and two-way set associative caches, but fades at higher associativities. Such conflicts are not unusual in array-based applications, especially when there are multiple grids in use at once. In Barnes and LU, the increase in processor count has little effect on the miss rate, sometimes causing a slight increase and sometimes causing a slight decrease.

Increasing the cache size usually has a beneficial effect on performance, since it reduces the frequency of costly cache misses. Figure I.9 illustrates the change in miss rate as cache size is increased for 16 processors, showing the portion of the miss rate due to coherence misses and to uniprocessor capacity misses. Two effects can lead to a miss rate that does not decrease—at least not as quickly as we might expect—as cache size increases: inherent communication and plateaus in the miss rate. Inherent communication leads to a certain frequency of coherence misses that are not significantly affected by increasing cache size. Thus, if the cache size is increased while maintaining a fixed problem size, the coherence miss rate



**Figure I.8 Data miss rates can vary in nonobvious ways as the processor count is increased from 1 to 16.** The miss rates include both coherence and capacity miss rates. The compulsory misses in these benchmarks are all very small and are included in the capacity misses. Most of the misses in these applications are generated by accesses to data that are potentially shared, although in the applications with larger miss rates (FFT and Ocean), it is the capacity misses rather than the coherence misses that comprise the majority of the miss rate. Data are potentially shared if they are allocated in a portion of the address space used for shared data. In all except Ocean, the potentially shared data are heavily shared, while in Ocean only the boundaries of the subgrids are actually shared, although the entire grid is treated as a potentially shared data object. Of course, since the boundaries change as we increase the processor count (for a fixed-size problem), different amounts of the grid become shared. The anomalous increase in capacity miss rate for Ocean in moving from 1 to 2 processors arises because of conflict misses in accessing the subgrids. In all cases except Ocean, the fraction of the cache misses caused by coherence transactions rises when a fixed-size problem is run on an increasing number of processors. In Ocean, the coherence misses initially fall as we add processors due to a large number of misses that are write ownership misses to data that are potentially, but not actually, shared. As the subgrids begin to fit in the aggregate cache (around 16 processors), this effect lessens. The single-processor numbers include write upgrade misses, which occur in this protocol even if the data are not actually shared, since they are in the shared state. For all these runs, the cache size is 64 KB, two-way set associative, with 32-byte blocks. Notice that the scale on the y-axis for each benchmark is different, so that the behavior of the individual benchmarks can be seen clearly.



**Figure I.9** The miss rate usually drops as the cache size is increased, although coherence misses dampen the effect. The block size is 32 bytes and the cache is two-way set associative. The processor count is fixed at 16 processors. Observe that the scale for each graph is different.

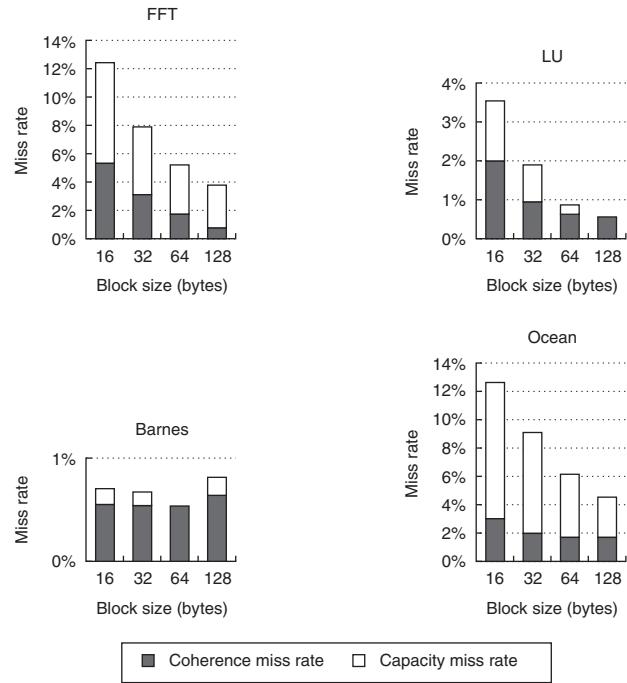
eventually limits the decrease in cache miss rate. This effect is most obvious in Barnes, where the coherence miss rate essentially becomes the entire miss rate.

A less important effect is a temporary plateau in the capacity miss rate that arises when the application has some fraction of its data present in cache but some significant portion of the dataset does not fit in the cache or in caches that are slightly bigger. In LU, a very small cache (about 4 KB) can capture the pair of  $16 \times 16$  blocks used in the inner loop; beyond that, the next big improvement in capacity miss rate occurs when both matrices fit in the caches, which occurs when the total cache size is between 4 MB and 8 MB. This effect, sometimes called a *working set effect*, is partly at work between 32 KB and 128 KB for FFT, where the capacity miss rate drops only 0.3%. Beyond that cache size, a faster decrease in the capacity miss rate is seen, as a major data structure begins to reside in the cache. These plateaus are common in programs that deal with large arrays in a structured fashion.

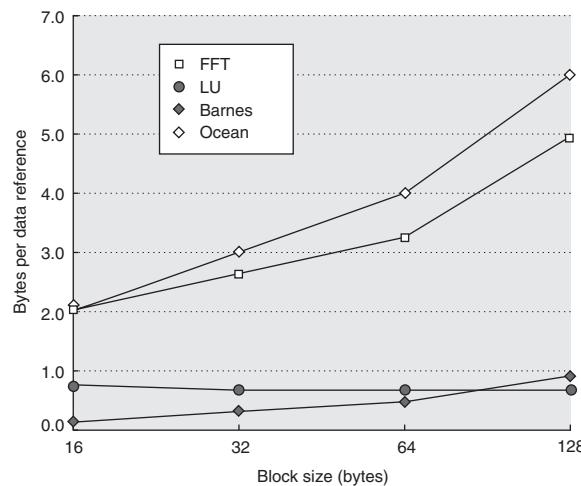
Increasing the block size is another way to change the miss rate in a cache. In uniprocessors, larger block sizes are often optimal with larger caches. In

multiprocessors, two new effects come into play: a reduction in spatial locality for shared data and a potential increase in miss rate due to false sharing. Several studies have shown that shared data have lower spatial locality than unshared data. Poorer locality means that, for shared data, fetching larger blocks is less effective than in a uniprocessor because the probability is higher that the block will be replaced before all its contents are referenced. Likewise, increasing the basic size also increases the potential frequency of false sharing, increasing the miss rate.

Figure I.10 shows the miss rates as the cache block size is increased for a 16-processor run with a 64 KB cache. The most interesting behavior is in Barnes, where the miss rate initially declines and then rises due to an increase in the number of coherence misses, which probably occurs because of false sharing. In the other benchmarks, increasing the block size decreases the overall miss rate. In Ocean and LU, the block size increase affects both the coherence and capacity miss rates about equally. In FFT, the coherence miss rate is actually decreased at a faster rate than the capacity miss rate. This reduction occurs because the communication in FFT is structured to be very efficient. In less optimized programs, we would expect more



**Figure I.10** The data miss rate drops as the cache block size is increased. All these results are for a 16-processor run with a 64 KB cache and two-way set associativity. Once again we use different scales for each benchmark.



**Figure I.11 Bus traffic for data misses climbs steadily as the block size in the data cache is increased.** The factor of 3 increase in traffic for Ocean is the best argument against larger block sizes. Remember that our protocol treats ownership or upgrade misses the same as other misses, slightly increasing the penalty for large cache blocks; in both Ocean and FFT, this simplification accounts for less than 10% of the traffic.

false sharing and less spatial locality for shared data, resulting in more behavior like that of Barnes.

Although the drop in miss rates with longer blocks may lead you to believe that choosing a longer block size is the best decision, the bottleneck in bus-based multiprocessors is often the limited memory and bus bandwidth. Larger blocks mean more bytes on the bus per miss. Figure I.11 shows the growth in bus traffic as the block size is increased. This growth is most serious in the programs that have a high miss rate, especially Ocean. The growth in traffic can actually lead to performance slowdowns due both to longer miss penalties and to increased bus contention.

### Performance of a Scientific Workload on a Distributed-Memory Multiprocessor

The performance of a directory-based multiprocessor depends on many of the same factors that influence the performance of bus-based multiprocessors (e.g., cache size, processor count, and block size), as well as the distribution of misses to various locations in the memory hierarchy. The location of a requested data item depends on both the initial allocation and the sharing patterns. We start by examining the basic cache performance of our scientific/technical workload and then look at the effect of different types of misses.

Because the multiprocessor is larger and has longer latencies than our snooping-based multiprocessor, we begin with a slightly larger cache (128 KB) and a larger block size of 64 bytes.

In distributed-memory architectures, the distribution of memory requests between local and remote is key to performance because it affects both the consumption of global bandwidth and the latency seen by requests. Therefore, for the figures in this section, we separate the cache misses into local and remote requests. In looking at the figures, keep in mind that, for these applications, most of the remote misses that arise are coherence misses, although some capacity misses can also be remote, and in some applications with poor data distribution such misses can be significant.

As Figure I.12 shows, the miss rates with these cache sizes are not affected much by changes in processor count, with the exception of Ocean, where the miss rate rises at 64 processors. This rise results from two factors: an increase in mapping conflicts in the cache that occur when the grid becomes small, which leads to a rise in local misses, and an increase in the number of the coherence misses, which are all remote.

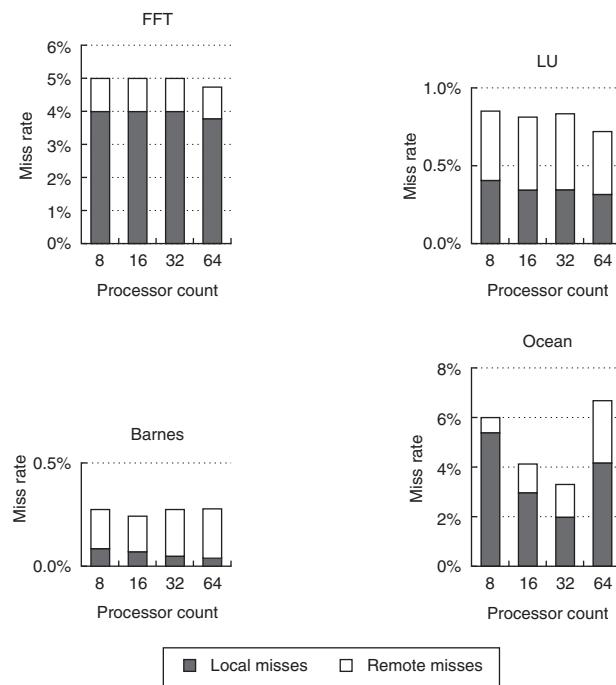
Figure I.13 shows how the miss rates change as the cache size is increased, assuming a 64-processor execution and 64-byte blocks. These miss rates decrease at rates that we might expect, although the dampening effect caused by little or no reduction in coherence misses leads to a slower decrease in the remote misses than in the local misses. By the time we reach the largest cache size shown, 512 KB, the remote miss rate is equal to or greater than the local miss rate. Larger caches would amplify this trend.

We examine the effect of changing the block size in Figure I.14. Because these applications have good spatial locality, increases in block size reduce the miss rate, even for large blocks, although the performance benefits for going to the largest blocks are small. Furthermore, most of the improvement in miss rate comes from a reduction in the local misses.

Rather than plot the memory traffic, Figure I.15 plots the number of bytes required per data reference versus block size, breaking the requirement into local and global bandwidth. In the case of a bus, we can simply aggregate the demands of each processor to find the total demand for bus and memory bandwidth. For a scalable interconnect, we can use the data in Figure I.15 to compute the required per-node global bandwidth and the estimated bisection bandwidth, as the next example shows.

---

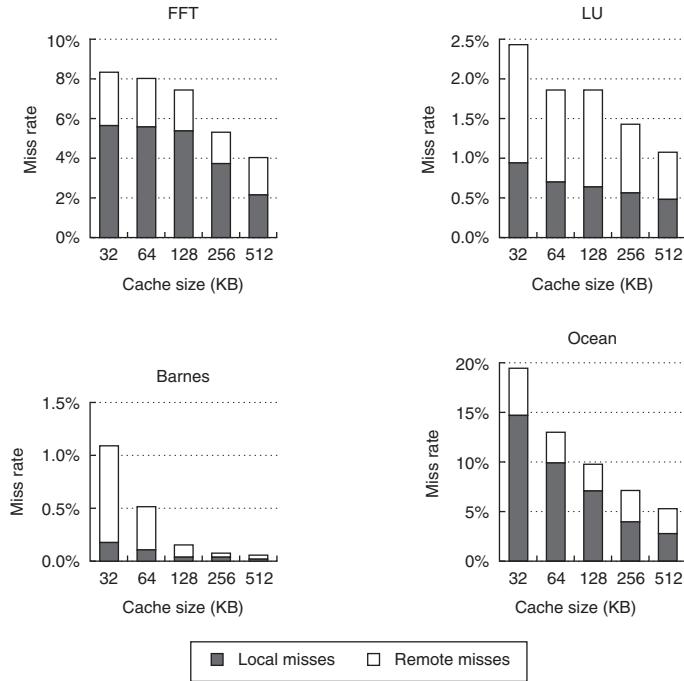
**Example** Assume a 64-processor multiprocessor with 1 GHz processors that sustain one memory reference per processor clock. For a 64-byte block size, the remote miss rate is 0.7%. Find the per-node and estimated bisection bandwidth for FFT. Assume that the processor does not stall for remote memory requests; this might be true if, for example, all remote data were prefetched. How do these bandwidth requirements compare to various interconnection technologies?



**Figure I.12** The data miss rate is often steady as processors are added for these benchmarks. Because of its grid structure, Ocean has an initially decreasing miss rate, which rises when there are 64 processors. For Ocean, the local miss rate drops from 5% at 8 processors to 2% at 32, before rising to 4% at 64. The remote miss rate in Ocean, driven primarily by communication, rises monotonically from 1% to 2.5%. Note that, to show the detailed behavior of each benchmark, different scales are used on the y-axis. The cache for all these runs is 128 KB, two-way set associative, with 64-byte blocks. Remote misses include any misses that require communication with another node, whether to fetch the data or to deliver an invalidate. In particular, in this figure and other data in this section, the measurement of remote misses includes write upgrade misses where the data are up to date in the local memory but cached elsewhere and, therefore, require invalidations to be sent. Such invalidations do indeed generate remote traffic, but may or may not delay the write, depending on the consistency model.

FFT performs all-to-all communication, so the bisection bandwidth is equal to the number of processors times the per-node bandwidth, or about  $64 \times 448 \text{ MB/sec} = 28.7 \text{ GB/sec}$ . The SGI Origin 3000 with 64 processors has a bisection bandwidth of about 50 GB/sec. No standard networking technology comes close.

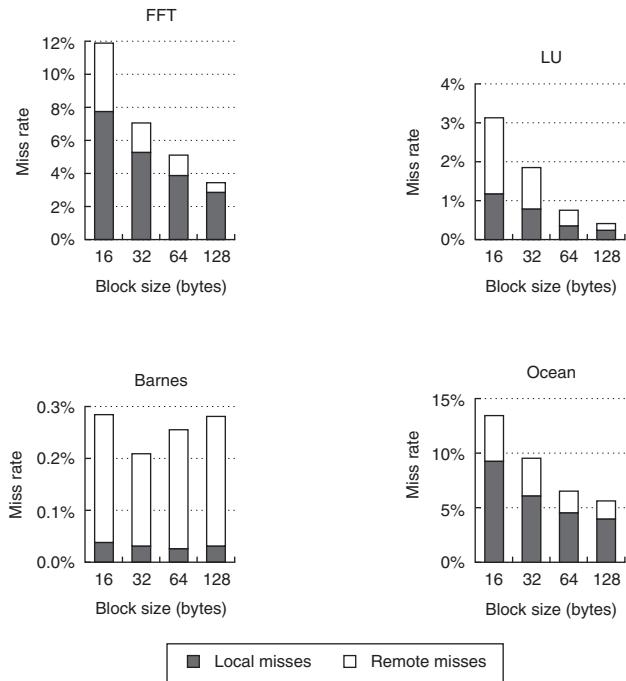
**Answer** The per-node bandwidth is simply the number of data bytes per reference times the reference rate:  $0.7\% \times 1 \text{ GB/sec} \times 64 = 448 \text{ MB/sec}$ . This rate is somewhat higher than the hardware sustainable transfer rate for the CrayT3E (using a block prefetch)



**Figure I.13 Miss rates decrease as cache sizes grow.** Steady decreases are seen in the local miss rate, while the remote miss rate declines to varying degrees, depending on whether the remote miss rate had a large capacity component or was driven primarily by communication misses. In all cases, the decrease in the local miss rate is larger than the decrease in the remote miss rate. The plateau in the miss rate of FFT, which we mentioned in the last section, ends once the cache exceeds 128 KB. These runs were done with 64 processors and 64-byte cache blocks.

and lower than that for an SGI Origin 3000 (1.6 GB/processor pair). The FFT per-node bandwidth demand exceeds the bandwidth sustainable from the fastest standard networks by more than a factor of 5.

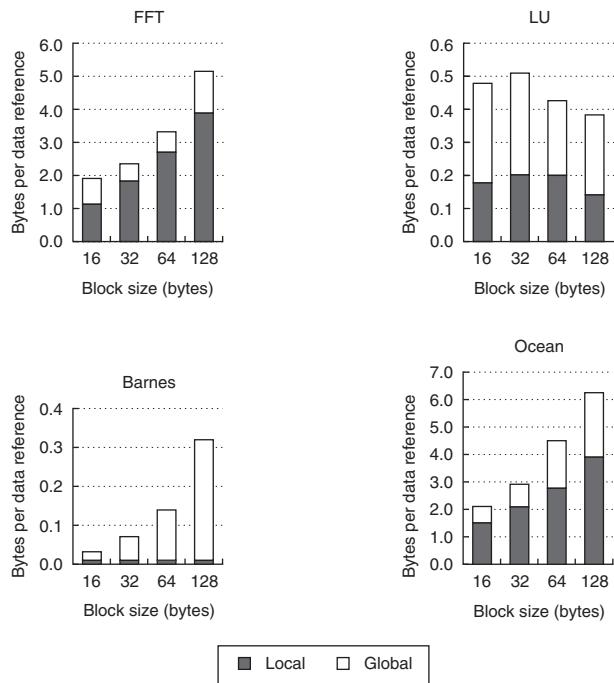
The previous example looked at the bandwidth demands. The other key issue for a parallel program is remote memory access time, or latency. To get insight into this, we use a simple example of a directory-based multiprocessor. Figure I.16 shows the parameters we assume for our simple multiprocessor model. It assumes that the time to first word for a local memory access is 85 processor cycles and that the path to local memory is 16 bytes wide, while the network interconnect is 4 bytes wide. This model ignores the effects of contention, which are probably not too serious in the parallel benchmarks we examine, with the possible exception of FFT, which uses all-to-all communication. Contention could have a serious performance impact in other workloads.



**Figure I.14** Data miss rate versus block size assuming a 128 KB cache and 64 processors in total. Although difficult to see, the coherence miss rate in Barnes actually rises for the largest block size, just as in the last section.

Figure I.17 shows the cost in cycles for the average memory reference, assuming the parameters in Figure I.16. Only the latencies for each reference type are counted. Each bar indicates the contribution from cache hits, local misses, remote misses, and three-hop remote misses. The cost is influenced by the total frequency of cache misses and upgrades, as well as by the distribution of the location where the miss is satisfied. The cost for a remote memory reference is fairly steady as the processor count is increased, except for Ocean. The increasing miss rate in Ocean for 64 processors is clear in Figure I.12. As the miss rate increases, we should expect the time spent on memory references to increase also.

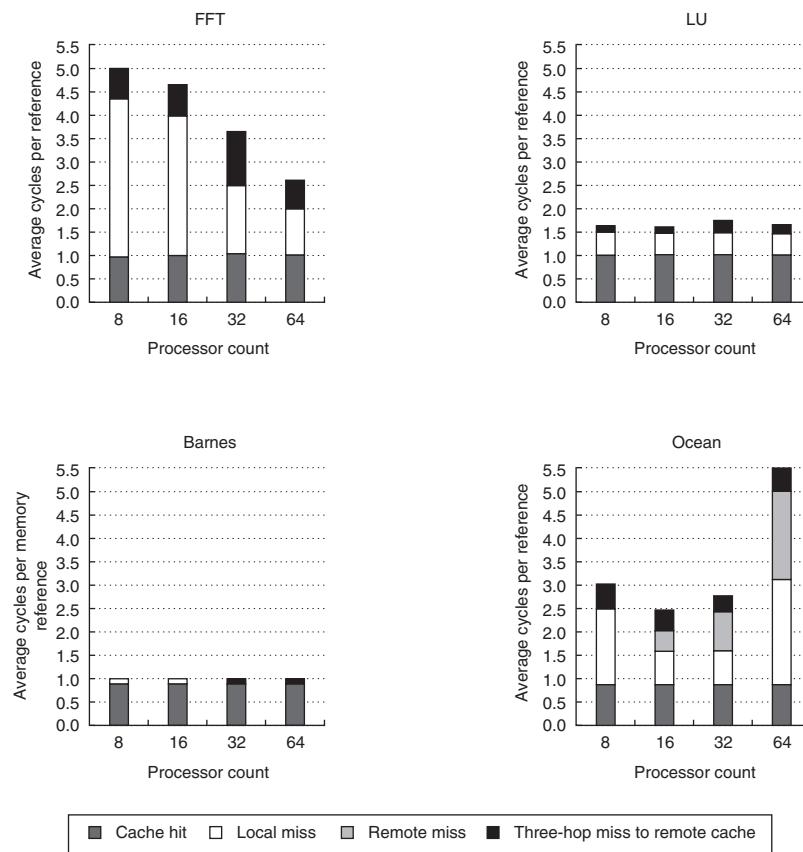
Although Figure I.17 shows the memory access cost, which is the dominant multiprocessor cost in these benchmarks, a complete performance model would need to consider the effect of contention in the memory system, as well as the losses arising from synchronization delays.



**Figure I.15** The number of bytes per data reference climbs steadily as block size is increased. These data can be used to determine the bandwidth required per node both internally and globally. The data assume a 128 KB cache for each of 64 processors.

| Characteristic                                      | Processor clock cycles<br>≤16 processors | Processor clock cycles<br>17–64 processors |
|-----------------------------------------------------|------------------------------------------|--------------------------------------------|
| Cache hit                                           | 1                                        | 1                                          |
| Cache miss to local memory                          | 85                                       | 85                                         |
| Cache miss to remote home directory                 | 125                                      | 150                                        |
| Cache miss to remotely cached data (three-hop miss) | 140                                      | 170                                        |

**Figure I.16 Characteristics of the example directory-based multiprocessor.** Misses can be serviced locally (including from the local directory), at a remote home node, or using the services of both the home node and another remote node that is caching an exclusive copy. This last case is called a three-hop miss and has a higher cost because it requires interrogating both the home directory and a remote cache. Note that this simple model does not account for invalidation time but does include some factor for increasing interconnect time. These remote access latencies are based on those in an SGI Origin 3000, the fastest scalable interconnect system in 2001, and assume a 500 MHz processor.



**Figure I.17** The effective latency of memory references in a DSM multiprocessor depends both on the relative frequency of cache misses and on the location of the memory where the accesses are served. These plots show the memory access cost (a metric called average memory access time in Chapter 2) for each of the benchmarks for 8, 16, 32, and 64 processors, assuming a 512 KB data cache that is two-way set associative with 64-byte blocks. The average memory access cost is composed of four different types of accesses, with the cost of each type given in Figure I.16. For the Barnes and LU benchmarks, the low miss rates lead to low overall access times. In FFT, the higher access cost is determined by a higher local miss rate (1–4%) and a significant three-hop miss rate (1%). The improvement in FFT comes from the reduction in local miss rate from 4% to 1%, as the aggregate cache increases. Ocean shows the biggest change in the cost of memory accesses, and the highest overall cost at 64 processors. The high cost is driven primarily by a high local miss rate (average 1.6%). The memory access cost drops from 8 to 16 processors as the grids more easily fit in the individual caches. At 64 processors, the dataset size is too small to map properly and both local misses and coherence misses rise, as we saw in Figure I.12.

## I.6

## Performance Measurement of Parallel Processors with Scientific Applications

One of the most controversial issues in parallel processing has been how to measure the performance of parallel processors. Of course, the straightforward answer is to measure a benchmark as supplied and to examine wall-clock time. Measuring wall-clock time obviously makes sense; in a parallel processor, measuring CPU time can be misleading because the processors may be idle but unavailable for other uses.

Users and designers are often interested in knowing not just how well a multiprocessor performs with a certain fixed number of processors, but also how the performance scales as more processors are added. In many cases, it makes sense to scale the application or benchmark, since if the benchmark is unscaled, effects arising from limited parallelism and increases in communication can lead to results that are pessimistic when the expectation is that more processors will be used to solve larger problems. Thus, it is often useful to measure the speedup as processors are added both for a fixed-size problem and for a scaled version of the problem, providing an unscaled and a scaled version of the speedup curves. The choice of how to measure the uniprocessor algorithm is also important to avoid anomalous results, since using the parallel version of the benchmark may underestimate the uniprocessor performance and thus overstate the speedup.

Once we have decided to measure scaled speedup, the question is *how* to scale the application. Let's assume that we have determined that running a benchmark of size  $n$  on  $p$  processors makes sense. The question is how to scale the benchmark to run on  $m \times p$  processors. There are two obvious ways to scale the problem: (1) keeping the amount of memory used per processor constant, and (2) keeping the total execution time, assuming perfect speedup, constant. The first method, called *memory-constrained scaling*, specifies running a problem of size  $m \times n$  on  $m \times p$  processors. The second method, called *time-constrained scaling*, requires that we know the relationship between the running time and the problem size, since the former is kept constant. For example, suppose the running time of the application with data size  $n$  on  $p$  processors is proportional to  $n^2/p$ . Then, with time-constrained scaling, the problem to run is the problem whose ideal running time on  $m \times p$  processors is still  $n^2/p$ . The problem with this ideal running time has size  $\sqrt{m} \times n$ .

---

**Example** Suppose we have a problem whose execution time for a problem of size  $n$  is proportional to  $n^3$ . Suppose the actual running time on a 10-processor multiprocessor is 1 hour. Under the time-constrained and memory-constrained scaling models, find the size of the problem to run and the effective running time for a 100-processor multiprocessor.

**Answer** For the time-constrained problem, the ideal running time is the same, 1 hour, so the problem size is  $\sqrt[3]{10} \times n$  or 2.15 times larger than the original. For

memory-constrained scaling, the size of the problem is  $10n$  and the ideal execution time is  $10^3/10$ , or 100 hours! Since most users will be reluctant to run a problem on an order of magnitude more processors for 100 times longer, this size problem is probably unrealistic.

In addition to the scaling methodology, there are questions as to how the program should be scaled when increasing the problem size affects the quality of the result. Often, we must change other application parameters to deal with this effect. As a simple example, consider the effect of time to convergence for solving a differential equation. This time typically increases as the problem size increases, since, for example, we often require more iterations for the larger problem. Thus, when we increase the problem size, the total running time may scale faster than the basic algorithmic scaling would indicate.

For example, suppose that the number of iterations grows as the log of the problem size. Then, for a problem whose algorithmic running time is linear in the size of the problem, the effective running time actually grows proportional to  $n \log n$ . If we scaled from a problem of size  $m$  on 10 processors, purely algorithmic scaling would allow us to run a problem of size  $10m$  on 100 processors. Accounting for the increase in iterations means that a problem of size  $k \times m$ , where  $k \log k = 10$ , will have the same running time on 100 processors. This problem size yields a scaling of  $5.72m$ , rather than  $10m$ .

In practice, scaling to deal with error requires a good understanding of the application and may involve other factors, such as error tolerances (for example, it affects the cell-opening criteria in Barnes-Hut). In turn, such effects often significantly affect the communication or parallelism properties of the application as well as the choice of problem size.

Scaled speedup is not the same as unscaled (or true) speedup; confusing the two has led to erroneous claims (e.g., see the discussion in Section I.6). Scaled speedup has an important role, but only when the scaling methodology is sound and the results are clearly reported as using a scaled version of the application. Singh, Hennessy, and Gupta [1993] described these issues in detail.

## I.7

### Implementing Cache Coherence

In this section, we explore the challenge of implementing cache coherence, starting first by dealing with the challenges in a snooping coherence protocol, which we simply alluded to in Chapter 5. Implementing a directory protocol adds some additional complexity to a snooping protocol, primarily arising from the absence of broadcast, which forces the use of a different mechanism to resolve races. Furthermore, the larger processor count of a directory-based multiprocessor means that we cannot retain assumptions of unlimited buffering and must find new ways to avoid deadlock. Let's start with the snooping protocols.

As we mentioned in Chapter 5, the challenge of implementing misses in a snooping coherence protocol without a bus lies in finding a way to make the multi-step miss process appear atomic. Both an upgrade miss and a write miss require the same basic processing and generate the same implementation challenges; for simplicity, we focus on upgrade misses. Here are the steps in handling an upgrade miss:

1. Detect the miss and compose an invalidate message for transmission to other caches.
2. When access to the broadcast communication link is available, transmit the message.
3. When the invalidates have been processed, the processor updates the state of the cache block and then proceeds with the write that caused the upgrade miss.

There are two related difficulties that can arise. First, how will two processors, P1 and P2, that attempt to upgrade the same cache block at the same time resolve the race? Second, when at step 3, how does a processor know when all invalidates have been processed so that it can complete the step?

The solution to finding a winner in the race lies in the ordering imposed by the broadcast communication medium. The communication medium must broadcast any cache miss to all the nodes. If P1 and P2 attempt to broadcast at the same time, we must ensure that either P1's message will reach P2 first or P2's will reach P1 first. This property will be true if there is a single channel through which all ingoing and outgoing requests from a node must pass through and if the communication network does not accept a message unless it can guarantee delivery (i.e., it is effectively circuit switched, see Appendix F). If both P1 and P2 initiate their attempts to broadcast an invalidate simultaneously, then the network can accept only one of these operations and delay the other. This ordering ensures that either P1 or P2 will complete its communication in step 2 first. The network can explicitly signal when it accepts a message and can guarantee it will be the next transmission; alternatively, a processor can simply watch the network for its own request, knowing that once the request is seen, it will be fully transmitted to all processors before any subsequent messages.

Now, suppose P1 wins the race to transmit its invalidate; once it knows it has won the race, it can continue with step 3 and complete the miss handling. There is a potential problem, however, for P2. When P2 undertook step 1, it believed that the block was in the shared state, but for P1 to advance at step 3, it must know that P2 has processed the invalidate, which must change the state of the block at P2 to invalid! One simple solution is for P2 to notice that it has lost the race, by observing that P1's invalidate is broadcast before its own invalidate. P2 can then invalidate the block and generate a write miss to get the data. P1 will see its invalidate before P2's, so it will change the block to modified and update the data, which guarantees forward progress and avoids deadlock. When P1 sees the subsequent invalidate to a block in the Modified state (a possibility that cannot arise in our basic protocol discussed in Chapter 5), it knows that it was the winner of a race. It can simply

ignore the invalidate, knowing that it will be followed by a write miss, or it can write the block back to memory and make its state invalid.

Another solution is to give precedence to incoming requests over outgoing requests, so that before P2 can transmit its invalidate it must handle any pending invalidates or write misses. If any of those misses are for blocks with the same address as a pending outgoing message, the processor must be prepared to restart the write operation, since the incoming request may cause the state of the block to change. Notice that P1 knows that the invalidates will be processed once it has successfully completed the broadcast, since precedence is given to invalidate messages over outgoing requests. (Because it does not employ broadcast, a processor using a directory protocol cannot know when an invalidate is received; instead, explicit acknowledgments are required, as we discuss in the next section. Indeed, as we will see, it cannot even know it has won the race to become the owner until its request is acknowledged.)

Reads will also require a multiple-step process, since we need to get the data back from memory or a remote cache (in a write-back cache system), but reads do not introduce fundamentally new problems beyond what exists for writes.

There are, however, a few additional tricky edge cases that must be handled correctly. For example, in a write-back cache, a processor can generate a read miss that requires a write-back, which it could delay, while giving the read miss priority. If a snoop request appears for the cache block that is to be written back, the processor must discover this and send the data back. Failure to do so can create a deadlock situation. A similar tricky situation exists when a processor generates a write miss, which will make a block exclusive, but, before the processor receives the data and can update the block, other processors generate read misses for that block. The read misses cannot be processed until the writing processor receives the data and updates the block.

One of the more difficult problems occurs in a write-back cache where the data for a read or write miss can come either from memory or from one of the processor caches, but the requesting processor will not know *a priori* where the data will come from. In most bus-based systems, a single global signal is used to indicate whether any processor has the exclusive (and hence the most up-to-date) copy; otherwise, the memory responds. These schemes can work with a pipelined interconnection by requiring that processors signal whether they have the exclusive copy within a fixed number of cycles after the miss is broadcast.

In a modern multiprocessor, however, it is essentially impossible to bound the amount of time required for a snoop request to be processed. Instead, a mechanism is required to determine whether the memory has an up-to-date copy. One solution is to add coherence bits to the memory, indicating whether the data are exclusive in a remote cache. This mechanism begins to move toward the directory approach, whose implementation challenges we consider next.

### Implementing Cache Coherence in a DSM Multiprocessor

Implementing a directory-based cache coherence protocol requires overcoming all the problems related to nonatomic actions for a snooping protocol without

the use of broadcast (see Chapter 5), which forced a serialization on competing writes and also ensured the serialization required for the memory consistency model. Avoiding the need to broadcast is a central goal for a directory-based system, so another method for ensuring serialization is necessary.

The serialization of requests for exclusive access to a memory block is easily enforced since those requests will be serialized when they reach the unique directory for the specified block. If the directory controller simply ensures that one request is completely serviced before the next is begun, writes will be serialized. Because the requesters cannot know ahead of time who will win the race and because the communication is not a broadcast, the directory must signal to the winner when it completes the processing of the winner's request. This is done by a message that supplies the data on a write miss or by an explicit acknowledgment message that grants ownership in response to an invalidation request.

What about the loser in this race? The simplest solution is for the system to send a *negative acknowledge*, or *NAK*, which requires that the requesting node regenerate its request. (This is the equivalent of a collision in the broadcast network in a snooping scheme, which requires that one of the transmitting nodes retry its communication.) We will see in the next section why the NAK approach, as opposed to buffering the request, is attractive.

Although the acknowledgment that a requesting node has ownership is completed when the write miss or ownership acknowledgment message is transmitted, we still do not know that the invalidates have been received and processed by the nodes that were in the sharing set. All memory consistency models eventually require (either before the next cache miss or at a synchronization point, for example) that a processor knows that all the invalidates for a write have been processed. In a snooping scheme, the nature of the broadcast network provides this assurance.

How can we know when the invalidates are complete in a directory scheme? The only way to know that the invalidates have been completed is to have the destination nodes of the invalidate messages (the members of the sharing set) explicitly acknowledge the invalidation messages sent from the directory. Who should they be acknowledged to? There are two possibilities. In the first the acknowledgments can be sent to the directory, which can count them, and when all acknowledgments have been received, confirm this with a single message to the original requester. Alternatively, when granting ownership, the directory can tell the register how many acknowledgments to expect. The destinations of the invalidate messages can then send an acknowledgment directly to the requester, whose identity is provided by the directory. Most existing implementations use the latter scheme, since it reduces the possibility of creating a bottleneck at a directory. Although the requirement for acknowledgments is an additional complexity in directory protocols, this requirement arises from the avoidance of a serialization mechanism, such as the snooping broadcast operation, which in itself is the limit to scalability.

## Avoiding Deadlock from Limited Buffering

A new complication in the implementation is introduced by the potential scale of a directory-based multiprocessor. In Chapter 5, we assumed that the network could always accept a coherence message and that the request would be acted upon at some point. In a much larger multiprocessor, this assumption of unlimited buffering may be unreasonable. What happens when the network does not have unlimited buffering? The major implication of this limit is that a cache or directory controller may be unable to complete a message send. This could lead to deadlock.

The potential deadlock arises from three properties, which characterize many deadlock situations:

1. More than one resource is needed to complete a transaction: Message buffers are needed to generate requests, create replies and acknowledgments, and accept replies.
2. Resources are held until a nonatomic transaction completes: The buffer used to create the reply cannot be freed until the reply is accepted, for reasons we will see shortly.
3. There is no global partial order on the acquisition of resources: Nodes can generate requests and replies at will.

These characteristics lead to deadlock, and avoiding deadlock requires breaking one of these properties. Freeing up resources without completing a transaction is difficult, since the transaction must be completely backed out and cannot be left half-finished. Hence, our approach will be to try to resolve the need for multiple resources. We cannot simply eliminate this need, but we can try to ensure that the resources will always be available.

One way to ensure that a transaction can always complete is to guarantee that there are always buffers to accept messages. Although this is possible for a small multiprocessor with processors that block on a cache miss or have a small number of outstanding misses, it may not be very practical in a directory protocol, since a single write could generate many invalidate messages. In addition, features such as prefetch and multiple outstanding misses increase the amount of buffering required. There is an alternative strategy, which most systems use and which ensures that a transaction will not actually be initiated until we can guarantee that it has the resources to complete. The strategy has four parts:

1. A separate network (physical or virtual) is used for requests and replies, where a reply is any message that a controller waits for in transitioning between states. This ensures that new requests cannot block replies that will free up buffers.
2. Every request that expects a reply allocates space to accept the reply when the request is generated. If no space is available, the request waits. This ensures that a node can always accept a reply message, which will allow the replying node to free its buffer.

3. Any controller can reject with a NAK any request, but it can never NAK a reply. This prevents a transaction from starting if the controller cannot guarantee that it has buffer space for the reply.
4. Any request that receives a NAK in response is simply retried.

To see that there are no deadlocks with the four properties above, we must ensure that all replies can be accepted and that every request is eventually serviced. Since a cache controller or directory controller always allocates a buffer to handle the reply before issuing a request, it can always accept the reply when it returns. To see that every request is eventually serviced, we need only show that any request could be completed. Since every request starts with a read or write miss at a cache, it is sufficient to show that any read or write miss is eventually serviced. Since the write miss case includes the actions for a read miss as a subset, we focus on showing the write misses are serviced. The simplest situation is when the block is uncached; since that case is subsumed by the case when the block is shared, we focus on the shared and exclusive cases. Let's consider the case where the block is shared:

- The CPU attempts to do a write and generates a write miss that is sent to the directory. For simplicity, we can assume that the processor is stalled. Although it may issue further requests, it should not issue a request for the same cache block until the first one is completed. Requests for independent blocks can be handled separately.
- The write miss is sent to the directory controller for this memory block. Note that although one cache controller handles all the requests for a given cache block, regardless of its memory contents, the directory controller handles requests for different blocks as independent events (assuming sufficient buffering, which is allocated before the directory issues any further messages on behalf of the request). The only conflict at the directory controller is when two requests arrive for the same block. The controller must wait for the first operation to be completed. It can simply NAK the second request or buffer it, but it should not service the second request for a given memory block until the first is completed.
- Now consider what happens at the directory controller: Suppose the write miss is the next thing to arrive at the directory controller. The controller sends out the invalidates, which can always be accepted after a limited delay by the cache controller. Note that one possibility is that the cache controller has an outstanding miss for the same block. This is the dual case to the snooping scheme, and we must once again break the tie by forcing the cache controller to accept and act on the directory request. Depending on the exact timing, this cache controller will either get the cache line later from the directory or will receive a NAK and have to restart the process.

The case where the block is exclusive is somewhat trickier. Our analysis begins when the write miss arrives at the directory controller for processing. There are two cases to consider:

- The directory controller sends a fetch/invalidate message to the processor where it arrives to find the block in the exclusive state. The cache controller sends a data write-back to the home directory and makes its state invalid. This reply arrives at the home directory controller, which can always accept the reply, since it preallocated the buffer. The directory controller sends back the data to the requesting processor, which can always accept the reply; after the cache is updated, the requesting cache controller notifies the processor.
- The directory controller sends a fetch/invalidate message to the node indicated as owner. When the message arrives at the owner node, it finds that this cache controller has taken a read or write miss that caused the block to be replaced. In this case, the cache controller has already sent the block to the home directory with a data write-back and made the data unavailable. Since this is exactly the effect of the fetch/invalidate message, the protocol operates correctly in this case as well.

We have shown that our coherence mechanism operates correctly when the cache and directory controller can accept requests for operation on cache blocks for which they have no outstanding operations in progress, when replies are always accepted, and when requests can be NAKed and forced to retry. Like the case of the snooping protocol, the cache controller must be able to break ties, and it always does so by favoring the instructions from the directory. The ability to NAK requests is what allows an implementation with finite buffering to avoid deadlock.

### Implementing the Directory Controller

To implement a cache coherence scheme, the cache controller must have the same abilities it needed in the snooping case, namely, the capability of handling requests for independent blocks while awaiting a response to a request from the local processor. The incoming requests are still processed in order, and each one is completed before beginning the next. Should a cache controller receive too many requests in a short period of time, it can NAK them, knowing that the directory will subsequently regenerate the request.

The directory must also be multithreaded and able to handle requests for multiple blocks independently. This situation is somewhat different than having the cache controller handle incoming requests for independent blocks, since the directory controller will need to begin processing one request while an earlier one is still underway. The directory controller cannot wait for one to complete before servicing the next request, since this could lead to deadlock. Instead, the directory controller must be *reentrant*; that is, it must be capable of suspending its execution

while waiting for a reply and accepting another transaction. The only place this must occur is in response to read or write misses, while waiting for a response from the owner. This leads to three important observations:

1. The state of the controller need only be saved and restored while either a fetch operation from a remote location or a fetch/invalidate is outstanding.
2. The implementation can bound the number of outstanding transactions being handled in the directory by simply NAKing read or write miss requests that could cause the number of outstanding requests to be exceeded.
3. If instead of returning the data through the directory, the owner node forwards the data directly to the requester (as well as returning it to the directory), we can eliminate the need for the directory to handle more than one outstanding request. This motivation, in addition to the reduction of latency, is the reason for using the forwarding style of protocol. There are other complexities from forwarding protocols that arise when requests arrive closely spaced in time.

The major remaining implementation difficulty is to handle NAKs. One alternative is for each processor to keep track of its outstanding transactions so it knows, when the NAK is received, what the requested transaction was. The alternative is to bundle the original request into the NAK, so that the controller receiving the NAK can determine what the original request was. Because every request allocates a slot to receive a reply and a NAK is a reply, NAKs can always be received. In fact, the buffer holding the return slot for the request can also hold information about the request, allowing the processor to reissue the request if it is NAKed.

In practice, great care is required to implement these protocols correctly and to avoid deadlock. The key ideas we have seen in this section—dealing with nonatomicity and finite buffering—are critical to ensuring a correct implementation. Designers have found that both formal and informal verification techniques are helpful for ensuring that implementations are correct.

## I.8

### **The Custom Cluster Approach: Blue Gene/L**

Blue Gene/L (BG/L) is a scalable message-passing supercomputer whose design offers unprecedented computing density as measured by compute power per watt. By focusing on power efficiency, BG/L also achieves unmatched throughput per cubic foot. High computing density, combined with cost-effective nodes and extensive support for RAS, allows BG/L to efficiently scale to very large processor counts.

BG/L is a distributed-memory, message-passing computer but one that is quite different from the cluster-based, often throughput-oriented computers that rely on commodity technology in the processors, interconnect, and, sometimes, the packaging and system-level organization. BG/L uses a special customized processing node that contains two processors (derived from low-power, lower-clock-rate PowerPC 440 chips used in the embedded market), caches, and interconnect logic.

A complete computing node is formed by adding SDRAM chips, which are the only commodity semiconductor parts in the BG/L design.

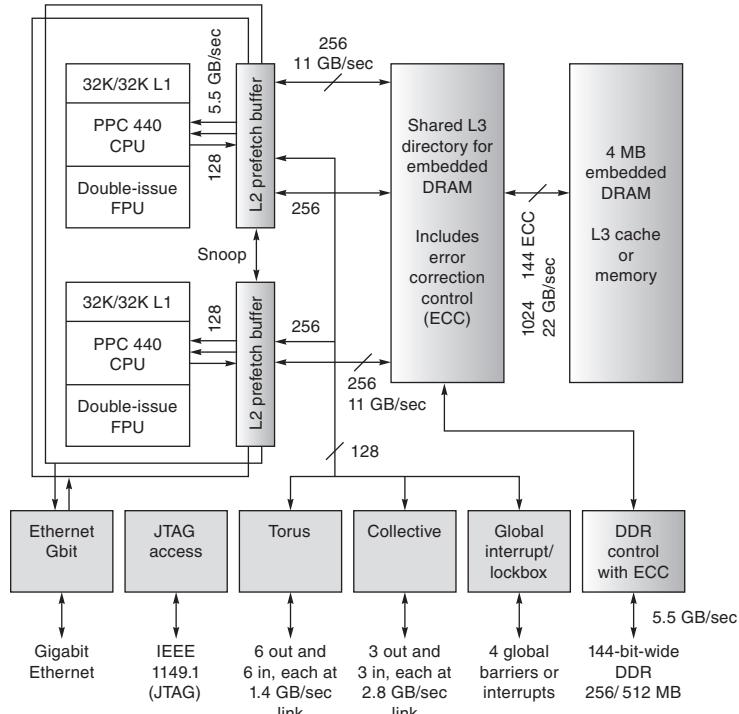
BG/L consists of up to 64 K nodes organized into 32 racks each containing 1 K nodes in about 50 cubic feet. Each rack contains two double-sided boards with 512 nodes each. Due to the high density within a board and rack, 85% of the interconnect is within a single rack, greatly reducing the complexity and latency associated with connections between racks. Furthermore, the compact size of a rack, which is enabled by the low power and high density of each node, greatly improves efficiency, since the interconnection network for connections within a single rack are integrated into the single compute chip that comprises each node.

Appendix F discusses the main BL/G interconnect network, which is a three-dimensional torus. There are four other networks: Gigabit Ethernet, connected at designated I/O nodes; a JTAG network used for test; a barrier network; and a global collective network. The barrier network contains four independent channels and can be used for performing a global *or* or a global *and* across all the processors with latency of less than 1.5 microseconds. The global collective network connects all the processors in a tree and is used for global operations. It supports a variety of integer reductions directly, avoiding the need to involve the processor, and leading to times for large-scale reductions that are 10 to 100 times faster than in typical supercomputers. The collective network can also be used to broadcast a single value efficiently. Support for the collective network as well as the torus is included in the chip that forms the heart of each processing node.

### *The Blue Gene/L Computing Node*

Each BG/L node consists of a single processing chip and several SDRAM chips. The BG/L processing chip, shown in Figure I.18, contains the following:

1. Two PowerPC 440 CPUs, each a two-issue superscalar with a seven-stage pipeline and speculative out-order issue capability, clocked at a modest (and power-saving) 700 MHz. Each CPU has separate 32 KB I and D caches that are nonblocking with up to four outstanding misses. Cache coherence must be enforced in software. Each CPU also contains a pair of floating-point coprocessors, each with its own FP register set and each capable of issuing a multiply-add each clock cycle, supporting a special SIMD instruction set capability that includes complex arithmetic using a pair of registers and 128-bit operands.
2. Separate fully associative L2 caches, each with 2 KB of data and a 128-byte block size, that act essentially like prefetch buffers. The L2 cache controllers recognize streamed data access and also handle prefetch from L3 or main memory. They have low latency (11 cycles) and provide high bandwidth (5 bytes per clock). The L2 prefetch buffer can supply 5.5 GB/sec to the L1 caches.
3. A 4 MB L3 cache implemented with embedded DRAM. Each L2 buffer is connected by a bus supplying 11 GB/sec of bandwidth from the L3 cache.



**Figure I.18** The BG/L processing node. The unfilled boxes are the PowerPC processors with added floating-point units. The solid gray boxes are network interfaces, and the shaded lighter gray boxes are part of the memory system, which is supplemented by DDR RAMS.

4. A memory bus supporting 256 to 512 MB of DDR DRAMS and providing 5.5 GB/sec of memory bandwidth to the L3 cache. This amount of memory might seem rather modest for each node, given that the node contains two processors, each with two FP units. Indeed Amdahl's rule of thumb (1 MB per 1 MIPS) and an assumption of 25% of peak performance would favor about 2.7 times the memory per node. For floating-point-intensive applications where the computational need usually grows faster than linear in the memory size, the upper limit of 512 MB/node is probably reasonable.
5. Support logic for the five interconnection networks.

By placing all the logic other than DRAMs into a single chip, BG/L achieves higher density, lower power, and lower cost, making it possible to pack the processing nodes extremely densely. The density in terms allows the interconnection networks to be low latency, high bandwidth, and quite cost effective. The combination yields a supercomputer that scales very cost-effectively, yielding an order-of-magnitude improvement



**Figure I.19** The 64 K-processor Blue Gene/L system.

in GFLOPs/watt over other approaches as well as significant improvements in GFLOPS/\$ for very large-scale multiprocessors.

For example, BG/L with 64 K nodes has a peak performance of 360 TF and uses about 1.4 megawatts. To achieve 360 TF peak using the Power5+, which is the most power-efficient, high-end FP processor, would require about 23,500 processors (the dual processor can execute up to 8 FLOPs/clock at 1.9 GHz). The power requirement for just the processors, without external cache, DRAM, or interconnect, would be about 2.9 megawatts, or about double the power of the entire BG/L system. Likewise, the smaller die size of the BG/L node and its need for DRAMs as the only external chip produce significant cost savings versus a node built using a high-end multiprocessor. Figure I.19 shows a photo of the 64K node BG/L. The total size occupied by this 128K-processor multiprocessor is comparable to that occupied by earlier multiprocessors with 16K processors.

## I.9

### Concluding Remarks

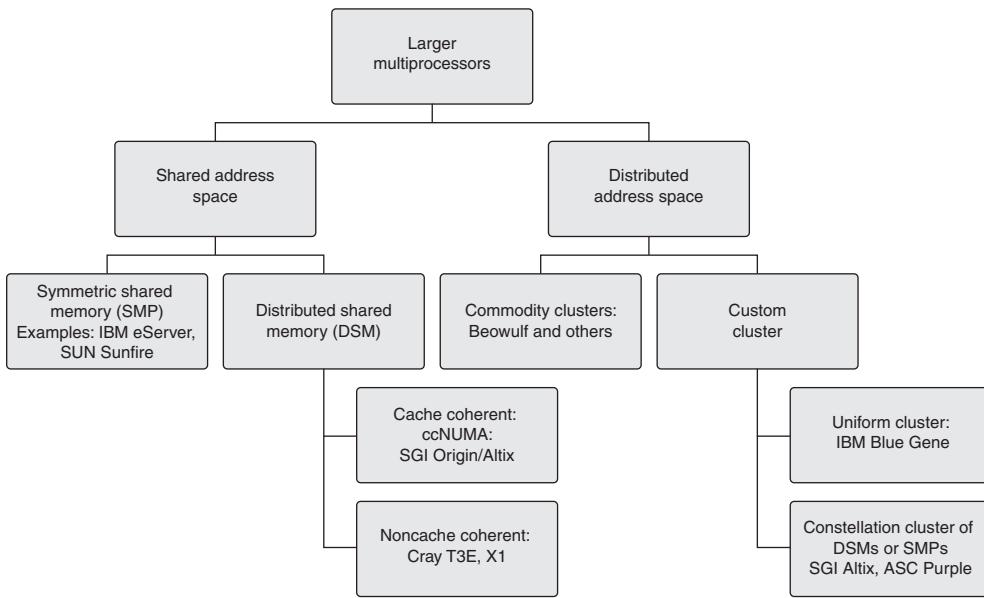
The landscape of large-scale multiprocessors has changed dramatically over the past five to ten years. While some form of clustering is now used for all the largest-scale multiprocessors, calling them all “clusters” ignores significant differences in architecture, implementation style, cost, and performance. Bell and Gray

| Terminology                     | Characteristics                                                                                                                                                                                                                                            | Examples                                              |
|---------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------|
| MPP                             | Originally referred to a class of architectures characterized by large numbers of small, typically custom processors and usually using an SIMD style architecture.                                                                                         | Connection Machines CM-2                              |
| SMP (symmetric multiprocessor)  | Shared-memory multiprocessors with a symmetric relationship to memory; also called UMA (uniform memory access). Scalable versions of these architectures used multistage interconnection networks, typically configured with at most 64 to 128 processors. | SUN Sunfire, NEC Earth Simulator                      |
| DSM (distributed shared memory) | A class of architectures that support scalable shared memory in a distributed fashion. These architectures are available both with and without cache coherence and typically can support hundreds to thousands of processors.                              | SGI Origin and Altix, Cray T3E, Cray X1, IBM p5 590/5 |
| Cluster                         | A class of multiprocessors using message passing. The individual nodes are either commodities or customized, likewise the interconnect.                                                                                                                    | See commodity and custom clusters                     |
| Commodity cluster               | A class of clusters where the nodes are truly commodities, typically headless workstations, motherboards, or blade servers, connected with a SAN or LAN usually accessible via an I/O bus.                                                                 | “Beowulf” and other “homemade” clusters               |
| Custom cluster                  | A cluster architecture where the nodes and the interconnect are customized and more tightly integrated than in a commodity cluster. Also called distributed memory or message passing multiprocessors.                                                     | IBM Blue Gene, Cray XT3                               |
| Constellation                   | Large-scale multiprocessors that use clustering of smaller-scale multiprocessors, typically with a DSM or SMP architecture and 32 or more processors.                                                                                                      | Larger SGI Origin/Altix, ASC Purple                   |

**Figure I.20 A classification of large-scale multiprocessors.** The term *MPP*, which had the original meaning described above, has been used more recently, and less precisely, to refer to all large-scale multiprocessors. None of the commercial shipping multiprocessors is a true MPP in the original sense of the word, but such an approach may make sense in the future. Both the SMP and DSM class includes multiprocessors with vector support. The term *constellation* has been used in different ways; the above usage seems both intuitive and precise [Dongarra et al. 2005].

[2002] discussed this trend, arguing that clusters will dominate. While Dongarra et al. [2005] agreed that some form of clustering is almost inevitable in the largest multiprocessors, they developed a more nuanced classification that attempts to distinguish among a variety of different approaches.

In Figure I.20 we summarize the range of terminology that has been used for large-scale multiprocessors and focus on defining the terms from an architectural and implementation perspective. Figure I.21 shows the hierarchical relationship of these different architecture approaches. Although there has been some convergence in architectural approaches over the past 15 years, the TOP500 list, which reports the 500 fastest computers in the world as measured by the Linpack benchmark, includes commodity clusters, customized clusters, Symmetric Multiprocessors (SMPs), DSMs, and constellations, as well as processors that are both scalar and vector.



**Figure I.21** The space of large-scale multiprocessors and the relation of different classes.

Nonetheless, there are some clearly emerging trends, which we can see by looking at the distribution of types of multiprocessors in the TOP500 list:

1. Clusters represent a majority of the systems. The lower development effort for clusters has clearly been a driving force in making them more popular. The high-end multiprocessor market has not grown sufficiently large to support full-scale, highly customized designs as the dominant choice.
2. The majority of the clusters are commodity clusters, often put together by users, rather than a system vendor designing a standard product.
3. Although commodity clusters dominate in their representation, the top 25 entries on the list are much more varied and include 9 custom clusters (primarily instances of Blue Gene or Cray XT3 systems), 2 constellations, 8 commodity clusters, 2 SMPs (one of which is the NEC Earth Simulator, which has nodes with vector processors), and 4 DSM multiprocessors.
4. Vector processors, which once dominated the list, have almost disappeared.
5. The IBM Blue Gene dominates the top 10 systems, showing the advantage of an approach that uses some commodity processor cores, but customizes many other functions and balances performance, power, and packaging density.
6. Architectural convergence has been driven more by market effects (lack of growth, limited suppliers, etc.) than by a clear-cut consensus on the best architectural approaches.

Software, both applications and programming languages and environments, remains the big challenge for parallel computing, just as it was 30 years ago, when multiprocessors such as the Illiac IV were being designed. The combination of ease of programming with high parallel performance remains elusive. Until better progress is made on this front, convergence toward a single programming model and underlying architectural approach (remembering that for uniprocessors we essentially have one programming model and one architectural approach!) will be slow or will be driven by factors other than proven architectural superiority.

---

|      |                                                 |      |
|------|-------------------------------------------------|------|
| J.1  | Introduction                                    | J-2  |
| J.2  | Basic Techniques of Integer Arithmetic          | J-2  |
| J.3  | Floating Point                                  | J-13 |
| J.4  | Floating-Point Multiplication                   | J-17 |
| J.5  | Floating-Point Addition                         | J-21 |
| J.6  | Division and Remainder                          | J-27 |
| J.7  | More on Floating-Point Arithmetic               | J-32 |
| J.8  | Speeding Up Integer Addition                    | J-37 |
| J.9  | Speeding Up Integer Multiplication and Division | J-44 |
| J.10 | Putting It All Together                         | J-57 |
| J.11 | Fallacies and Pitfalls                          | J-62 |
| J.12 | Historical Perspective and References           | J-63 |
|      | Exercises                                       | J-67 |

J

---

# Computer Arithmetic

**by David Goldberg  
Xerox Palo Alto Research Center**

The Fast drives out the Slow even if the Fast is wrong.

W. Kahan

**J.1****Introduction**

Although computer arithmetic is sometimes viewed as a specialized part of CPU design, it is a very important part. This was brought home for Intel in 1994 when their Pentium chip was discovered to have a bug in the divide algorithm. This floating-point flaw resulted in a flurry of bad publicity for Intel and also cost them a lot of money. Intel took a \$300 million write-off to cover the cost of replacing the buggy chips.

In this appendix, we will study some basic floating-point algorithms, including the division algorithm used on the Pentium. Although a tremendous variety of algorithms have been proposed for use in floating-point accelerators, actual implementations are usually based on refinements and variations of the few basic algorithms presented here. In addition to choosing algorithms for addition, subtraction, multiplication, and division, the computer architect must make other choices. What precisions should be implemented? How should exceptions be handled? This appendix will give you the background for making these and other decisions.

Our discussion of floating point will focus almost exclusively on the IEEE floating-point standard (IEEE 754) because of its rapidly increasing acceptance. Although floating-point arithmetic involves manipulating exponents and shifting fractions, the bulk of the time in floating-point operations is spent operating on fractions using integer algorithms (but not necessarily sharing the hardware that implements integer instructions). Thus, after our discussion of floating point, we will take a more detailed look at integer algorithms.

Some good references on computer arithmetic, in order from least to most detailed, are Chapter 3 of Patterson and Hennessy [2009]; Chapter 7 of Hamacher, Vranesic, and Zaky [1984]; Gosling [1980]; and Scott [1985].

**J.2****Basic Techniques of Integer Arithmetic**

Readers who have studied computer arithmetic before will find most of this section to be review.

**Ripple-Carry Addition**

Adders are usually implemented by combining multiple copies of simple components. The natural components for addition are *half adders* and *full adders*. The half adder takes two bits  $a$  and  $b$  as input and produces a sum bit  $s$  and a carry bit  $c_{\text{out}}$  as output. Mathematically,  $s = (a+b) \bmod 2$ , and  $c_{\text{out}} = \lfloor (a+b)/2 \rfloor$ , where  $\lfloor \cdot \rfloor$  is the floor function. As logic equations,  $s = \bar{a}\bar{b} + \bar{a}b$  and  $c_{\text{out}} = ab$ , where  $ab$  means  $a \wedge b$  and  $a+b$  means  $a \vee b$ . The half adder is also called a (2,2) adder, since it takes two inputs and produces two outputs. The full adder

is a (3,2) adder and is defined by  $s = (a + b + c) \bmod 2$ ,  $c_{\text{out}} = \lfloor (a + b + c)/2 \rfloor$ , or the logic equations

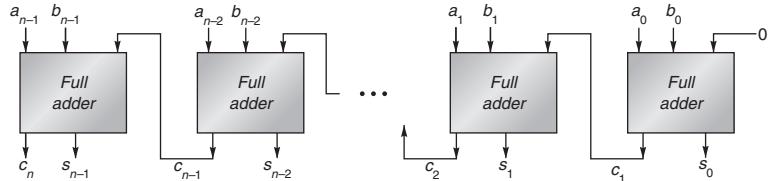
$$\text{J.2.1} \quad s = ab\bar{c} + \bar{a}b\bar{c} + \bar{a}\bar{b}c + abc$$

$$\text{J.2.2} \quad c_{\text{out}} = ab + ac + bc$$

The principal problem in constructing an adder for  $n$ -bit numbers out of smaller pieces is propagating the carries from one piece to the next. The most obvious way to solve this is with a *ripple-carry adder*, consisting of  $n$  full adders, as illustrated in Figure J.1. (In the figures in this appendix, the least-significant bit is always on the right.) The inputs to the adder are  $a_{n-1}a_{n-2}\dots a_0$  and  $b_{n-1}b_{n-2}\dots b_0$ , where  $a_{n-1}a_{n-2}\dots a_0$  represents the number  $a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \dots + a_0$ . The  $c_{i+1}$  output of the  $i$ th adder is fed into the  $c_{i+1}$  input of the next adder (the  $(i+1)$ -th adder) with the lower-order carry-in  $c_0$  set to 0. Since the low-order carry-in is wired to 0, the low-order adder could be a half adder. Later, however, we will see that setting the low-order carry-in bit to 1 is useful for performing subtraction.

In general, the time a circuit takes to produce an output is proportional to the maximum number of logic levels through which a signal travels. However, determining the exact relationship between logic levels and timings is highly technology dependent. Therefore, when comparing adders we will simply compare the number of logic levels in each one. How many levels are there for a ripple-carry adder? It takes two levels to compute  $c_1$  from  $a_0$  and  $b_0$ . Then it takes two more levels to compute  $c_2$  from  $c_1$ ,  $a_1$ ,  $b_1$ , and so on, up to  $c_n$ . So, there are a total of  $2n$  levels. Typical values of  $n$  are 32 for integer arithmetic and 53 for double-precision floating point. The ripple-carry adder is the slowest adder, but also the cheapest. It can be built with only  $n$  simple cells, connected in a simple, regular way.

Because the ripple-carry adder is relatively slow compared with the designs discussed in Section J.8, you might wonder why it is used at all. In technologies like CMOS, even though ripple adders take time  $O(n)$ , the constant factor is very small. In such cases short ripple adders are often used as building blocks in larger adders.



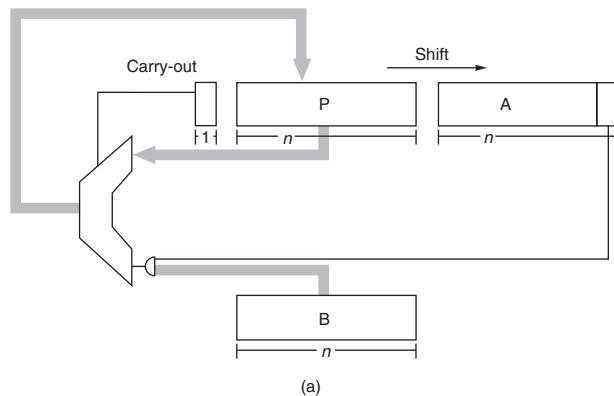
**Figure J.1 Ripple-carry adder, consisting of  $n$  full adders.** The carry-out of one full adder is connected to the carry-in of the adder for the next most-significant bit. The carries ripple from the least-significant bit (on the right) to the most-significant bit (on the left).

## Radix-2 Multiplication and Division

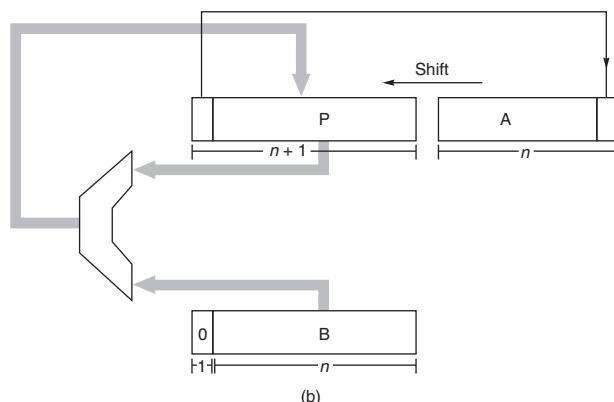
The simplest multiplier computes the product of two unsigned numbers, one bit at a time, as illustrated in Figure J.2(a). The numbers to be multiplied are  $a_{n-1}a_{n-2}\dots a_0$  and  $b_{n-1}b_{n-2}\dots b_0$ , and they are placed in registers A and B, respectively. Register P is initially 0. Each multiply step has two parts:

### Multiply Step

- (i) If the least-significant bit of A is 1, then register B, containing  $b_{n-1}b_{n-2}\dots b_0$ , is added to P; otherwise, 00…00 is added to P. The sum is placed back into P.
- (ii) Registers P and A are shifted right, with the carry-out of the sum being moved into the high-order bit of P, the low-order bit of P being moved into register A, and the rightmost bit of A, which is not used in the rest of the algorithm, being shifted out.



(a)



(b)

**Figure J.2** Block diagram of (a) multiplier and (b) divider for n-bit unsigned integers. Each multiplication step consists of adding the contents of P to either B or 0 (depending on the low-order bit of A), replacing P with the sum, and then shifting both P and A one bit right. Each division step involves first shifting P and A one bit left, subtracting B from P, and, if the difference is nonnegative, putting it into P. If the difference is nonnegative, the low-order bit of A is set to 1.

After  $n$  steps, the product appears in registers P and A, with A holding the lower-order bits.

The simplest divider also operates on unsigned numbers and produces the quotient bits one at a time. A hardware divider is shown in Figure J.2(b). To compute  $a/b$ , put  $a$  in the A register,  $b$  in the B register, and 0 in the P register and then perform  $n$  divide steps. Each divide step consists of four parts:

- Divide Step**
- (i) Shift the register pair (P,A) one bit left.
  - (ii) Subtract the content of register B (which is  $b_{n-1}b_{n-2}\dots b_0$ ) from register P, putting the result back into P.
  - (iii) If the result of step 2 is negative, set the low-order bit of A to 0, otherwise to 1.
  - (iv) If the result of step 2 is negative, restore the old value of P by adding the contents of register B back into P.

After repeating this process  $n$  times, the A register will contain the quotient, and the P register will contain the remainder. This algorithm is the binary version of the paper-and-pencil method; a numerical example is illustrated in Figure J.3(a).

Notice that the two block diagrams in Figure J.2 are very similar. The main difference is that the register pair (P,A) shifts right when multiplying and left when dividing. By allowing these registers to shift bidirectionally, the same hardware can be shared between multiplication and division.

The division algorithm illustrated in Figure J.3(a) is called *restoring*, because if subtraction by  $b$  yields a negative result, the P register is restored by adding  $b$  back in. The restoring algorithm has a variant that skips the restoring step and instead works with the resulting negative numbers. Each step of this *nonrestoring* algorithm has three parts:

**Nonrestoring** If P is negative,

- Divide Step** (i-a) Shift the register pair (P,A) one bit left.

- (ii-a) Add the contents of register B to P.

Else,

- (i-b) Shift the register pair (P,A) one bit left.

- (ii-b) Subtract the contents of register B from P.

- (iii) If P is negative, set the low-order bit of A to 0, otherwise set it to 1.

After repeating this  $n$  times, the quotient is in A. If P is nonnegative, it is the remainder. Otherwise, it needs to be restored (i.e., add  $b$ ), and then it will be the remainder. A numerical example is given in Figure J.3(b). Since steps (i-a) and (i-b) are the same, you might be tempted to perform this common step first, and then test the sign of P. That doesn't work, since the sign bit can be lost when shifting.

**J-6** ■ Appendix J *Computer Arithmetic*

| <b>P</b>      | <b>A</b>    |                                                                                |
|---------------|-------------|--------------------------------------------------------------------------------|
| 00000         | 1110        | Divide $14 = 1110_2$ by $3 = 11_2$ . B always contains $0011_2$ .              |
| 00001         | 110         | step 1(i): shift.                                                              |
| <u>-00011</u> |             | step 1(ii): subtract.                                                          |
| <u>-00010</u> | <b>1100</b> | step 1(iii): result is negative, set quotient bit to 0.                        |
| 00001         | <b>1100</b> | step 1(iv): restore.                                                           |
| 00011         | <b>100</b>  | step 2(i): shift.                                                              |
| <u>-00011</u> |             | step 2(ii): subtract.                                                          |
| 00000         | <b>1001</b> | step 2(iii): result is nonnegative, set quotient bit to 1.                     |
| 00001         | <b>001</b>  | step 3(i): shift.                                                              |
| <u>-00011</u> |             | step 3(ii): subtract.                                                          |
| <u>-00010</u> | <b>0010</b> | step 3(iii): result is negative, set quotient bit to 0.                        |
| 00001         | <b>0010</b> | step 3(iv): restore.                                                           |
| 00010         | <b>010</b>  | step 4(i): shift.                                                              |
| <u>-00011</u> |             | step 4(ii): subtract.                                                          |
| <u>-00001</u> | <b>0100</b> | step 4(iii): result is negative, set quotient bit to 0.                        |
| 00010         | <b>0100</b> | step 4(iv): restore. The quotient is $0100_2$ and the remainder is $00010_2$ . |

(a)

|               |             |                                                                   |
|---------------|-------------|-------------------------------------------------------------------|
| 00000         | 1110        | Divide $14 = 1110_2$ by $3 = 11_2$ . B always contains $0011_2$ . |
| 00001         | 110         | step 1(i-b): shift.                                               |
| <u>+11101</u> |             | step 1(ii-b): subtract b (add two's complement).                  |
| 11110         | <b>1100</b> | step 1(iii): P is negative, so set quotient bit to 0.             |
| 11101         | <b>100</b>  | step 2(i-a): shift.                                               |
| <u>+00011</u> |             | step 2(ii-a): add b.                                              |
| 00000         | <b>1001</b> | step 2(iii): P is nonnegative, so set quotient bit to 1.          |
| 00001         | <b>001</b>  | step 3(i-b): shift.                                               |
| <u>+11101</u> |             | step 3(ii-b): subtract b.                                         |
| 11110         | <b>0010</b> | step 3(iii): P is negative, so set quotient bit to 0.             |
| 11100         | <b>010</b>  | step 4(i-a): shift.                                               |
| <u>+00011</u> |             | step 4(ii-a): add b.                                              |
| 11111         | <b>0100</b> | step 4(iii): P is negative, so set quotient bit to 0.             |
| <u>+00011</u> |             | Remainder is negative, so do final restore step.                  |
| 00010         |             | The quotient is $0100_2$ and the remainder is $00010_2$ .         |

(b)

**Figure J.3** Numerical example of (a) restoring division and (b) nonrestoring division.

The explanation for why the nonrestoring algorithm works is this. Let  $r_k$  be the contents of the (P,A) register pair at step  $k$ , ignoring the quotient bits (which are simply sharing the unused bits of register A). In Figure J.3(a), initially A contains 14, so  $r_0 = 14$ . At the end of the first step,  $r_1 = 28$ , and so on. In the restoring algorithm, part (i) computes  $2r_k$  and then part (ii)  $2r_k - 2^n b$  ( $2^n b$  since  $b$  is subtracted from the left half). If  $2r_k - 2^n b \geq 0$ , both algorithms end the step with identical values in (P,A). If  $2r_k - 2^n b < 0$ , then the restoring algorithm restores this to  $2r_k$ , and the next step begins by computing  $r_{\text{res}} = 2(2r_k) - 2^n b$ . In the non-restoring algorithm,  $2r_k - 2^n b$  is kept as a negative number, and in the next step  $r_{\text{nonres}} = 2(2r_k - 2^n b) + 2^n b = 4r_k - 2^n b = r_{\text{res}}$ . Thus (P,A) has the same bits in both algorithms.

If  $a$  and  $b$  are unsigned  $n$ -bit numbers, hence in the range  $0 \leq a, b \leq 2^n - 1$ , then the multiplier in Figure J.2 will work if register P is  $n$  bits long. However, for division, P must be extended to  $n+1$  bits in order to detect the sign of P. Thus the adder must also have  $n+1$  bits.

Why would anyone implement restoring division, which uses the same hardware as nonrestoring division (the control is slightly different) but involves an extra addition? In fact, the usual implementation for restoring division doesn't actually perform an add in step (iv). Rather, the sign resulting from the subtraction is tested at the output of the adder, and only if the sum is nonnegative is it loaded back into the P register.

As a final point, before beginning to divide, the hardware must check to see whether the divisor is 0.

## Signed Numbers

There are four methods commonly used to represent signed  $n$ -bit numbers: *sign magnitude*, *two's complement*, *one's complement*, and *biased*. In the sign magnitude system, the high-order bit is the sign bit, and the low-order  $n-1$  bits are the magnitude of the number. In the two's complement system, a number and its negative add up to  $2^n$ . In one's complement, the negative of a number is obtained by complementing each bit (or, alternatively, the number and its negative add up to  $2^n - 1$ ). In each of these three systems, nonnegative numbers are represented in the usual way. In a biased system, nonnegative numbers do not have their usual representation. Instead, all numbers are represented by first adding them to the bias and then encoding this sum as an ordinary unsigned number. Thus, a negative number  $k$  can be encoded as long as  $k + \text{bias} \geq 0$ . A typical value for the bias is  $2^{n-1}$ .

**Example** Using 4-bit numbers ( $n=4$ ), if  $k=3$  (or in binary,  $k=0011_2$ ), how is  $-k$  expressed in each of these formats?

**Answer** In signed magnitude, the leftmost bit in  $k=0011_2$  is the sign bit, so flip it to 1:  $-k$  is represented by  $1011_2$ . In two's complement,  $k + 1101_2 = 2^4 = 16$ . So  $-k$  is represented by  $1101_2$ . In one's complement, the bits of  $k=0011_2$  are flipped, so  $-k$  is represented by  $1100_2$ . For a biased system, assuming a bias of  $2^{n-1}=8$ ,  $k$  is represented by  $k + \text{bias} = 1011_2$ , and  $-k$  by  $-k + \text{bias} = 0101_2$ .

The most widely used system for representing integers, two's complement, is the system we will use here. One reason for the popularity of two's complement is that it makes signed addition easy: Simply discard the carry-out from the highorder bit. To add  $5 + -2$ , for example, add  $0101_2$  and  $1110_2$  to obtain  $0011_2$ , resulting in the correct value of 3. A useful formula for the value of a two's complement number  $a_{n-1}a_{n-2}\cdots a_1a_0$  is

$$\text{J.2.3} \quad -a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \cdots + a_12^1 + a_0$$

As an illustration of this formula, the value of  $1101_2$  as a 4-bit two's complement number is  $-1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = -8 + 4 + 1 = -3$ , confirming the result of the example above.

*Overflow* occurs when the result of the operation does not fit in the representation being used. For example, if unsigned numbers are being represented using 4 bits, then  $6 = 0110_2$  and  $11 = 1011_2$ . Their sum (17) overflows because its binary equivalent ( $10001_2$ ) doesn't fit into 4 bits. For unsigned numbers, detecting overflow is easy; it occurs exactly when there is a carry-out of the most-significant bit. For two's complement, things are trickier: Overflow occurs exactly when the carry into the high-order bit is different from the (to be discarded) carry-out of the high-order bit. In the example of  $5 + -2$  above, a 1 is carried both into and out of the leftmost bit, avoiding overflow.

Negating a two's complement number involves complementing each bit and then adding 1. For instance, to negate  $0011_2$ , complement it to get  $1100_2$  and then add 1 to get  $1101_2$ . Thus, to implement  $a - b$  using an adder, simply feed  $a$  and  $\bar{b}$  (where  $\bar{b}$  is the number obtained by complementing each bit of  $b$ ) into the adder and set the low-order, carry-in bit to 1. This explains why the rightmost adder in Figure J.1 is a full adder.

Multiplying two's complement numbers is not quite as simple as adding them. The obvious approach is to convert both operands to be nonnegative, do an unsigned multiplication, and then (if the original operands were of opposite signs) negate the result. Although this is conceptually simple, it requires extra time and hardware. Here is a better approach: Suppose that we are multiplying  $a$  times  $b$  using the hardware shown in Figure J.2(a). Register A is loaded with the number  $a$ ; B is loaded with  $b$ . Since the content of register B is always  $b$ , we will use B and  $b$  interchangeably. If B is potentially negative but A is nonnegative, the only change needed to convert the unsigned multiplication algorithm into a two's complement one is to ensure that when P is shifted, it is shifted arithmetically; that is, the bit shifted into the high-order bit of P should be the sign bit of P (rather than the carry-out from the addition). Note that our  $n$ -bit-wide adder will now be adding  $n$ -bit two's complement numbers between  $-2^{n-1}$  and  $2^{n-1} - 1$ .

Next, suppose  $a$  is negative. The method for handling this case is called *Booth recoding*. Booth recoding is a very basic technique in computer arithmetic and will play a key role in Section J.9. The algorithm on page J-4 computes  $a \times b$  by examining the bits of  $a$  from least significant to most significant. For example, if  $a = 7 = 0111_2$ , then step (i) will successively add B, add B, add B, and add 0. Booth recoding “recodes” the number 7 as  $8 - 1 = 1000_2 - 0001_2 = 100\bar{1}$ , where  $\bar{1}$

represents  $-1$ . This gives an alternative way to compute  $a \times b$ , namely, successively subtract  $B$ , add 0, add 0, and add  $B$ . This is more complicated than the unsigned algorithm on page J-4, since it uses both addition and subtraction. The advantage shows up for negative values of  $a$ . With the proper recoding, we can treat  $a$  as though it were unsigned. For example, take  $a = -4 = 1100_2$ . Think of  $1100_2$  as the unsigned number 12, and recode it as  $12 = 16 - 4 = 10000_2 - 0100_2 = 10\bar{1}00$ . If the multiplication algorithm is only iterated  $n$  times ( $n=4$  in this case), the high-order digit is ignored, and we end up subtracting  $0100_2 = 4$  times the multiplier—exactly the right answer. This suggests that multiplying using a recoded form of  $a$  will work equally well for both positive and negative numbers. And, indeed, to deal with negative values of  $a$ , all that is required is to sometimes subtract  $b$  from  $P$ , instead of adding either  $b$  or 0 to  $P$ . Here are the precise rules: If the initial content of  $A$  is  $a_{n-1}\dots a_0$ , then at the  $i$ th multiply step the low-order bit of register  $A$  is  $a_i$ , and step (i) in the multiplication algorithm becomes:

- I. If  $a_i = 0$  and  $a_{i-1} = 0$ , then add 0 to  $P$ .
- II. If  $a_i = 0$  and  $a_{i-1} = 1$ , then add  $B$  to  $P$ .
- III. If  $a_i = 1$  and  $a_{i-1} = 0$ , then subtract  $B$  from  $P$ .
- IV. If  $a_i = 1$  and  $a_{i-1} = 1$ , then add 0 to  $P$ .

For the first step, when  $i=0$ , take  $a_{i-1}$  to be 0.

**Example** When multiplying  $-6$  times  $-5$ , what is the sequence of values in the  $(P,A)$  register pair?

**Answer** See Figure J.4.

| P      | A    |                                                                                                  |
|--------|------|--------------------------------------------------------------------------------------------------|
| 0000   | 1010 | Put $-6 = 1010_2$ into $A$ , $-5 = 1011_2$ into $B$ .                                            |
| 0000   | 1010 | step 1(i): $a_0 = a_{-1} = 0$ , so from rule I add 0.                                            |
| 0000   | 0101 | step 1(ii): shift.                                                                               |
| +0101  |      | step 2(i): $a_1 = 1$ , $a_0 = 0$ . Rule III says subtract $b$ (or add $-b = -1011_2 = 0101_2$ ). |
| 0101   | 0101 |                                                                                                  |
| 0010   | 1010 | step 2(ii): shift.                                                                               |
| + 1011 |      | step 3(i): $a_2 = 0$ , $a_1 = 1$ . Rule II says add $b$ (1011).                                  |
| 1101   | 1010 |                                                                                                  |
| 1110   | 1101 | step 3(ii): shift. (Arithmetic shift—load 1 into leftmost bit.)                                  |
| + 0101 |      | step 4(i): $a_3 = 1$ , $a_2 = 0$ . Rule III says subtract $b$ .                                  |
| 0011   | 1101 |                                                                                                  |
| 0001   | 1110 | step 4(ii): shift. Final result is $00011110_2 = 30$ .                                           |

**Figure J.4** Numerical example of Booth recoding. Multiplication of  $a = -6$  by  $b = -5$  to get 30.

The four prior cases can be restated as saying that in the  $i$ th step you should add  $(a_{i-1} - a_i)b$  to P. With this observation, it is easy to verify that these rules work, because the result of all the additions is

$$\sum_{i=0}^{n-1} b(a_{i-1} - a_i)2^i = b(-a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \dots + a_12 + a_0) + ba_{-1}$$

Using Equation J.2.3 (page J-8) together with  $a_{-1} = 0$ , the right-hand side is seen to be the value of  $b \times a$  as a two's complement number.

The simplest way to implement the rules for Booth recoding is to extend the A register one bit to the right so that this new bit will contain  $a_{i-1}$ . Unlike the naive method of inverting any negative operands, this technique doesn't require extra steps or any special casing for negative operands. It has only slightly more control logic. If the multiplier is being shared with a divider, there will already be the capability for subtracting  $b$ , rather than adding it. To summarize, a simple method for handling two's complement multiplication is to pay attention to the sign of P when shifting it right, and to save the most recently shifted-out bit of A to use in deciding whether to add or subtract  $b$  from P.

Booth recoding is usually the best method for designing multiplication hardware that operates on signed numbers. For hardware that doesn't directly implement it, however, performing Booth recoding in software or microcode is usually too slow because of the conditional tests and branches. If the hardware supports arithmetic shifts (so that negative  $b$  is handled correctly), then the following method can be used. Treat the multiplier  $a$  as if it were an unsigned number, and perform the first  $n - 1$  multiply steps using the algorithm on page J-4. If  $a < 0$  (in which case there will be a 1 in the low-order bit of the A register at this point), then subtract  $b$  from P; otherwise ( $a \geq 0$ ), neither add nor subtract. In either case, do a final shift (for a total of  $n$  shifts). This works because it amounts to multiplying  $b$  by  $-a_{n-1}2^{n-1} + \dots + a_12 + a_0$ , which is the value of  $a_{n-1}\dots a_0$  as a two's complement number by Equation J.2.3. If the hardware doesn't support arithmetic shift, then converting the operands to be nonnegative is probably the best approach.

Two final remarks: A good way to test a signed-multiply routine is to try  $-2^{n-1} \times -2^{n-1}$ , since this is the only case that produces a  $2n - 1$  bit result. Unlike multiplication, division is usually performed in hardware by converting the operands to be nonnegative and then doing an unsigned divide. Because division is substantially slower (and less frequent) than multiplication, the extra time used to manipulate the signs has less impact than it does on multiplication.

## Systems Issues

When designing an instruction set, a number of issues related to integer arithmetic need to be resolved. Several of them are discussed here.

First, what should be done about integer overflow? This situation is complicated by the fact that detecting overflow differs depending on whether the operands are signed or unsigned integers. Consider signed arithmetic first. There are three

approaches: Set a bit on overflow, trap on overflow, or do nothing on overflow. In the last case, software has to check whether or not an overflow occurred. The most convenient solution for the programmer is to have an enable bit. If this bit is turned on, then overflow causes a trap. If it is turned off, then overflow sets a bit (or, alternatively, have two different add instructions). The advantage of this approach is that both trapping and nontrapping operations require only one instruction. Furthermore, as we will see in Section J.7, this is analogous to how the IEEE floating-point standard handles floating-point overflow. Figure J.5 shows how some common machines treat overflow.

What about unsigned addition? Notice that none of the architectures in Figure J.5 traps on unsigned overflow. The reason for this is that the primary use of unsigned arithmetic is in manipulating addresses. It is convenient to be able to subtract from an unsigned address by adding. For example, when  $n=4$ , we can subtract 2 from the unsigned address  $10 = 1010_2$  by adding  $14 = 1110_2$ . This generates an overflow, but we would not want a trap to be generated.

A second issue concerns multiplication. Should the result of multiplying two  $n$ -bit numbers be a  $2n$ -bit result, or should multiplication just return the low-order  $n$  bits, signaling overflow if the result doesn't fit in  $n$  bits? An argument in favor of an  $n$ -bit result is that in virtually all high-level languages, multiplication is an operation in which arguments are integer variables and the result is an integer variable of the same type. Therefore, compilers won't generate code that utilizes a double-precision result. An argument in favor of a  $2n$ -bit result is that it can be used by an assembly language routine to substantially speed up multiplication of multiple-precision integers (by about a factor of 3).

A third issue concerns machines that want to execute one instruction every cycle. It is rarely practical to perform a multiplication or division in the same amount of time that an addition or register-register move takes. There are three possible approaches to this problem. The first is to have a single-cycle *multiply-step* instruction. This might do one step of the Booth algorithm. The second approach is to do integer multiplication in the floating-point unit and have it be part of the floating-point instruction set.

| Machine    | Trap on signed overflow?                                      | Trap on unsigned overflow? | Set bit on signed overflow?                                   | Set bit on unsigned overflow?      |
|------------|---------------------------------------------------------------|----------------------------|---------------------------------------------------------------|------------------------------------|
| VAX        | If enable is on                                               | No                         | Yes. Add sets V bit.                                          | Yes. Add sets C bit.               |
| IBM 370    | If enable is on                                               | No                         | Yes. Add sets cond code.                                      | Yes. Logical add sets cond code.   |
| Intel 8086 | No                                                            | No                         | Yes. Add sets V bit.                                          | Yes. Add sets C bit.               |
| MIPS R3000 | Two add instructions; one always traps, the other never does. | No                         | No. Software must deduce it from sign of operands and result. |                                    |
| SPARC      | No                                                            | No                         | Addcc sets V bit.<br>Add does not.                            | Addcc sets C bit.<br>Add does not. |

**Figure J.5 Summary of how various machines handle integer overflow.** Both the 8086 and SPARC have an instruction that traps if the V bit is set, so the cost of trapping on overflow is one extra instruction.

(This is what DLX does.) The third approach is to have an autonomous unit in the CPU do the multiplication. In this case, the result either can be guaranteed to be delivered in a fixed number of cycles—and the compiler charged with waiting the proper amount of time—or there can be an interlock. The same comments apply to division as well. As examples, the original SPARC had a multiply-step instruction but no divide-step instruction, while the MIPS R3000 has an autonomous unit that does multiplication and division (newer versions of the SPARC architecture added an integer multiply instruction). The designers of the HP Precision Architecture did an especially thorough job of analyzing the frequency of the operands for multiplication and division, and they based their multiply and divide steps accordingly. (See Magenheimer et al. [1988] for details.)

The final issue involves the computation of integer division and remainder for negative numbers. For example, what is  $-5 \text{ DIV } 3$  and  $-5 \text{ MOD } 3$ ? When computing  $x \text{ DIV } y$  and  $x \text{ MOD } y$ , negative values of  $x$  occur frequently enough to be worth some careful consideration. (On the other hand, negative values of  $y$  are quite rare.) If there are built-in hardware instructions for these operations, they should correspond to what high-level languages specify. Unfortunately, there is no agreement among existing programming languages. See Figure J.6.

One definition for these expressions stands out as clearly superior, namely,  $x \text{ DIV } y = \lfloor x/y \rfloor$ , so that  $5 \text{ DIV } 3 = 1$  and  $-5 \text{ DIV } 3 = -2$ . And MOD should satisfy  $x = (x \text{ DIV } y) \times y + x \text{ MOD } y$ , so that  $x \text{ MOD } y \geq 0$ . Thus,  $5 \text{ MOD } 3 = 2$ , and  $-5 \text{ MOD } 3 = 1$ . Some of the many advantages of this definition are as follows:

1. A calculation to compute an index into a hash table of size  $N$  can use MOD  $N$  and be guaranteed to produce a valid index in the range from 0 to  $N - 1$ .
2. In graphics, when converting from one coordinate system to another, there is no “glitch” near 0. For example, to convert from a value  $x$  expressed in a system that uses 100 dots per inch to a value  $y$  on a bitmapped display with 70 dots per inch, the formula  $y = (70 \times x) \text{ DIV } 100$  maps one or two  $x$  coordinates into each  $y$  coordinate. But if DIV were defined as in Pascal to be  $x/y$  rounded to 0, then 0 would have three different points  $(-1, 0, 1)$  mapped into it.
3.  $x \text{ MOD } 2^k$  is the same as performing a bitwise AND with a mask of  $k$  bits, and  $x \text{ DIV } 2^k$  is the same as doing a  $k$ -bit arithmetic right shift.

| Language | Division                 | Remainder                                           |
|----------|--------------------------|-----------------------------------------------------|
| FORTRAN  | $-5/3 = -1$              | $\text{MOD}(-5, 3) = -2$                            |
| Pascal   | $-5 \text{ DIV } 3 = -1$ | $-5 \text{ MOD } 3 = 1$                             |
| Ada      | $-5/3 = -1$              | $-5 \text{ MOD } 3 = 1$<br>$-5 \text{ REM } 3 = -2$ |
| C        | $-5/3$ undefined         | $-5\% 3$ undefined                                  |
| Modula-3 | $-5 \text{ DIV } 3 = -2$ | $-5 \text{ MOD } 3 = 1$                             |

**Figure J.6** Examples of integer division and integer remainder in various programming languages.

Finally, a potential pitfall worth mentioning concerns multiple-precision addition. Many instruction sets offer a variant of the `add` instruction that adds three operands: two  $n$ -bit numbers together with a third single-bit number. This third number is the carry from the previous addition. Since the multiple-precision number will typically be stored in an array, it is important to be able to increment the array pointer without destroying the carry bit.

### J.3

## Floating Point

Many applications require numbers that aren't integers. There are a number of ways that nonintegers can be represented. One is to use *fixed point*; that is, use integer arithmetic and simply imagine the binary point somewhere other than just to the right of the least-significant digit. Adding two such numbers can be done with an integer `add`, whereas multiplication requires some extra shifting. Other representations that have been proposed involve storing the logarithm of a number and doing multiplication by adding the logarithms, or using a pair of integers  $(a,b)$  to represent the fraction  $a/b$ . However, only one noninteger representation has gained widespread use, and that is *floating point*. In this system, a computer word is divided into two parts, an exponent and a significand. As an example, an exponent of  $-3$  and a significand of  $1.5$  might represent the number  $1.5 \times 2^{-3} = 0.1875$ . The advantages of standardizing a particular representation are obvious. Numerical analysts can build up high-quality software libraries, computer designers can develop techniques for implementing high-performance hardware, and hardware vendors can build standard accelerators. Given the predominance of the floating-point representation, it appears unlikely that any other representation will come into widespread use.

The semantics of floating-point instructions are not as clear-cut as the semantics of the rest of the instruction set, and in the past the behavior of floating-point operations varied considerably from one computer family to the next. The variations involved such things as the number of bits allocated to the exponent and significand, the range of exponents, how rounding was carried out, and the actions taken on exceptional conditions like underflow and overflow. Computer architecture books used to dispense advice on how to deal with all these details, but fortunately this is no longer necessary. That's because the computer industry is rapidly converging on the format specified by IEEE standard 754-1985 (also an international standard, IEC 559). The advantages of using a standard variant of floating point are similar to those for using floating point over other noninteger representations.

IEEE arithmetic differs from many previous arithmetics in the following major ways:

1. When rounding a "halfway" result to the nearest floating-point number, it picks the one that is even.
2. It includes the *special values* NaN,  $\infty$ , and  $-\infty$ .

3. It uses *denormal* numbers to represent the result of computations whose value is less than  $1.0 \times 2^{E_{\min}}$ .
4. It rounds to nearest by default, but it also has three other rounding modes.
5. It has sophisticated facilities for handling exceptions.

To elaborate on (1), note that when operating on two floating-point numbers, the result is usually a number that cannot be exactly represented as another floating-point number. For example, in a floating-point system using base 10 and two significant digits,  $6.1 \times 0.5 = 3.05$ . This needs to be rounded to two digits. Should it be rounded to 3.0 or 3.1? In the IEEE standard, such halfway cases are rounded to the number whose low-order digit is even. That is, 3.05 rounds to 3.0, not 3.1. The standard actually has four *rounding modes*. The default is *round to nearest*, which rounds ties to an even number as just explained. The other modes are round toward 0, round toward  $+\infty$ , and round toward  $-\infty$ .

We will elaborate on the other differences in following sections. For further reading, see IEEE [1985], Cody et al. [1984], and Goldberg [1991].

## Special Values and Denormals

Probably the most notable feature of the standard is that by default a computation continues in the face of exceptional conditions, such as dividing by 0 or taking the square root of a negative number. For example, the result of taking the square root of a negative number is a *NaN* (*Not a Number*), a bit pattern that does not represent an ordinary number. As an example of how NaNs might be useful, consider the code for a zero finder that takes a function  $F$  as an argument and evaluates  $F$  at various points to determine a zero for it. If the zero finder accidentally probes outside the valid values for  $F$ , then  $F$  may well cause an exception. Writing a zero finder that deals with this case is highly language and operating-system dependent, because it relies on how the operating system reacts to exceptions and how this reaction is mapped back into the programming language. In IEEE arithmetic it is easy to write a zero finder that handles this situation and runs on many different systems. After each evaluation of  $F$ , it simply checks to see whether  $F$  has returned a NaN; if so, it knows it has probed outside the domain of  $F$ .

In IEEE arithmetic, if the input to an operation is a NaN, the output is NaN (e.g.,  $3 + \text{NaN} = \text{NaN}$ ). Because of this rule, writing floating-point subroutines that can accept NaN as an argument rarely requires any special case checks. For example, suppose that  $\arccos$  is computed in terms of  $\arctan$ , using the formula  $\arccos x = 2 \arctan(\sqrt{(1-x)/(1+x)})$ . If  $\arctan$  handles an argument of NaN properly,  $\arccos$  will automatically do so, too. That's because if  $x$  is a NaN,  $1+x$ ,  $1-x$ ,  $(1+x)/(1-x)$ , and  $\sqrt{(1-x)/(1+x)}$  will also be NaNs. No checking for NaNs is required.

While the result of  $\sqrt{-1}$  is a NaN, the result of  $1/0$  is not a NaN, but  $+\infty$ , which is another special value. The standard defines arithmetic on infinities (there are

both  $+\infty$  and  $-\infty$ ) using rules such as  $1/\infty=0$ . The formula  $\arccos x = 2 \arctan(\sqrt{(1-x)/(1+x)})$  illustrates how infinity arithmetic can be used. Since  $\arctan x$  asymptotically approaches  $\pi/2$  as  $x$  approaches  $\infty$ , it is natural to define  $\arctan(\infty)=\pi/2$ , in which case  $\arccos(-1)$  will automatically be computed correctly as  $2 \arctan(\infty)=\pi$ .

The final kind of special values in the standard are *denormal* numbers. In many floating-point systems, if  $E_{\min}$  is the smallest exponent, a number less than  $1.0 \times 2^{E_{\min}}$  cannot be represented, and a floating-point operation that results in a number less than this is simply flushed to 0. In the IEEE standard, on the other hand, numbers less than  $1.0 \times 2^{E_{\min}}$  are represented using significands less than 1. This is called *gradual underflow*. Thus, as numbers decrease in magnitude below  $2^{E_{\min}}$ , they gradually lose their significance and are only represented by 0 when all their significance has been shifted out. For example, in base 10 with four significant figures, let  $x = 1.234 \times 10^{E_{\min}}$ . Then,  $x/10$  will be rounded to  $0.123 \times 10^{E_{\min}}$ , having lost a digit of precision. Similarly  $x/100$  rounds to  $0.012 \times 10^{E_{\min}}$ , and  $x/1000$  to  $0.001 \times 10^{E_{\min}}$ , while  $x/10000$  is finally small enough to be rounded to 0. Denormals make dealing with small numbers more predictable by maintaining familiar properties such as  $x=y \Leftrightarrow x-y=0$ . For example, in a flush-to-zero system (again in base 10 with four significant digits), if  $x = 1.256 \times 10^{E_{\min}}$  and  $y = 1.234 \times 10^{E_{\min}}$ , then  $x-y = 0.022 \times 10^{E_{\min}}$ , which flushes to zero. So even though  $x \neq y$ , the computed value of  $x-y=0$ . This never happens with gradual underflow. In this example,  $x-y = 0.022 \times 10^{E_{\min}}$  is a denormal number, and so the computation of  $x-y$  is exact.

## Representation of Floating-Point Numbers

Let us consider how to represent single-precision numbers in IEEE arithmetic. Single-precision numbers are stored in 32 bits: 1 for the sign, 8 for the exponent, and 23 for the fraction. The exponent is a signed number represented using the bias method (see the subsection “Signed Numbers,” page J-7) with a bias of 127. The term *biased exponent* refers to the unsigned number contained in bits 1 through 8, and *unbiased exponent* (or just exponent) means the actual power to which 2 is to be raised. The fraction represents a number less than 1, but the *significand* of the floating-point number is 1 plus the fraction part. In other words, if  $e$  is the biased exponent (value of the exponent field) and  $f$  is the value of the fraction field, the number being represented is  $1.f \times 2^{e-127}$ .

**Example** What single-precision number does the following 32-bit word represent?

1 10000001 01000000000000000000000000000000

**Answer** Considered as an unsigned number, the exponent field is 129, making the value of the exponent  $129 - 127 = 2$ . The fraction part is  $.01_2 = .25$ , making the significand 1.25. Thus, this bit pattern represents the number  $-1.25 \times 2^2 = -5$ .

The fractional part of a floating-point number (.25 in the example above) must not be confused with the significand, which is 1 plus the fractional part. The leading 1 in the significand  $1.f$  does not appear in the representation; that is, the leading bit is implicit. When performing arithmetic on IEEE format numbers, the fraction part is usually *unpacked*, which is to say the implicit 1 is made explicit.

Figure J.7 summarizes the parameters for single (and other) precisions. It shows the exponents for single precision to range from  $-126$  to  $127$ ; accordingly, the biased exponents range from  $1$  to  $254$ . The biased exponents of  $0$  and  $255$  are used to represent special values. This is summarized in Figure J.8. When the biased exponent is  $255$ , a zero fraction field represents infinity, and a nonzero fraction field represents a NaN. Thus, there is an entire family of NaNs. When the biased exponent and the fraction field are  $0$ , then the number represented is  $0$ . Because of the implicit leading 1, ordinary numbers always have a significand greater than or equal to  $1$ . Thus, a special convention such as this is required to represent  $0$ . Denormalized numbers are implemented by having a word with a zero exponent field represent the number  $0.f \times 2^{E_{\min}}$ .

The primary reason why the IEEE standard, like most other floating-point formats, uses biased exponents is that it means nonnegative numbers are ordered in the same way as integers. That is, the magnitude of floating-point numbers can be compared using an integer comparator. Another (related) advantage is that  $0$  is represented by a word of all 0s. The downside of biased exponents is that adding them is slightly awkward, because it requires that the bias be subtracted from their sum.

|                         | Single | Single extended | Double  | Double extended |
|-------------------------|--------|-----------------|---------|-----------------|
| $p$ (bits of precision) | 24     | $\geq 32$       | 53      | $\geq 64$       |
| $E_{\max}$              | 127    | $\geq 1023$     | 1023    | $\geq 16383$    |
| $E_{\min}$              | $-126$ | $\leq -1022$    | $-1022$ | $\leq -16382$   |
| Exponent bias           | 127    |                 | 1023    |                 |

**Figure J.7 Format parameters for the IEEE 754 floating-point standard.** The first row gives the number of bits in the significand. The blanks are unspecified parameters.

| Exponent                        | Fraction   | Represents                |
|---------------------------------|------------|---------------------------|
| $e = E_{\min} - 1$              | $f = 0$    | $\pm 0$                   |
| $e = E_{\min} - 1$              | $f \neq 0$ | $0.f \times 2^{E_{\min}}$ |
| $E_{\min} \leq e \leq E_{\max}$ | —          | $1.f \times 2^e$          |
| $e = E_{\max} + 1$              | $f = 0$    | $\pm \infty$              |
| $e = E_{\max} + 1$              | $f \neq 0$ | NaN                       |

**Figure J.8 Representation of special values.** When the exponent of a number falls outside the range  $E_{\min} \leq e \leq E_{\max}$ , then that number has a special interpretation as indicated in the table.

**J.4****Floating-Point Multiplication**

The simplest floating-point operation is multiplication, so we discuss it first. A binary floating-point number  $x$  is represented as a significand and an exponent,  $x=s \times 2^e$ . The formula

$$(s_1 \times 2^{e1}) \bullet (s_2 \times 2^{e2}) = (s_1 \bullet s_2) \times 2^{e1+e2}$$

shows that a floating-point multiply algorithm has several parts. The first part multiplies the significands using ordinary integer multiplication. Because floating-point numbers are stored in sign magnitude form, the multiplier need only deal with unsigned numbers (although we have seen that Booth recoding handles signed two's complement numbers painlessly). The second part rounds the result. If the significands are unsigned  $p$ -bit numbers (e.g.,  $p=24$  for single precision), then the product can have as many as  $2p$  bits and must be rounded to a  $p$ -bit number. The third part computes the new exponent. Because exponents are stored with a bias, this involves subtracting the bias from the sum of the biased exponents.

**Example** How does the multiplication of the single-precision numbers

$$1\ 1000001\ 0000\dots = -1 \times 2^3$$

$$0\ 1000001\ 1000\dots = 1 \times 2^4$$

proceed in binary?

**Answer** When unpacked, the significands are both 1.0, their product is 1.0, and so the result is of the form:

$$1\ ??????? 000\dots$$

To compute the exponent, use the formula:

$$\text{biased exp}(e_1 + e_2) = \text{biased exp}(e_1) + \text{biased exp}(e_2) - \text{bias}$$

From Figure J.7, the bias is  $127 = 01111111_2$ , so in two's complement  $-127$  is  $10000001_2$ . Thus, the biased exponent of the product is

$$\begin{array}{r} 10000010 \\ 10000011 \\ + 10000001 \\ \hline 10000110 \end{array}$$

Since this is 134 decimal, it represents an exponent of  $134 - \text{bias} = 134 - 127$ , as expected.

The interesting part of floating-point multiplication is rounding. Some of the different cases that can occur are illustrated in Figure J.9. Since the cases are similar in all bases, the figure uses human-friendly base 10, rather than base 2.

|     |                |                                                                                  |
|-----|----------------|----------------------------------------------------------------------------------|
| (a) | 1.23           |                                                                                  |
|     | $\times 6.78$  |                                                                                  |
|     | <u>8.3394</u>  | $r=9 > 5$ so round up<br>rounds to 8.34                                          |
|     | ↑              |                                                                                  |
| (b) | 2.83           |                                                                                  |
|     | $\times 4.47$  |                                                                                  |
|     | <u>12.6501</u> | $r=5$ and a following digit $\neq 0$ so round up<br>rounds to $1.27 \times 10^1$ |
|     | ↑              |                                                                                  |
| (c) | 1.28           |                                                                                  |
|     | $\times 7.81$  |                                                                                  |
|     | <u>09.9968</u> | $r=6 > 5$ so round up<br>rounds to $1.00 \times 10^1$                            |
|     | ↑              |                                                                                  |

**Figure J.9 Examples of rounding a multiplication.** Using base 10 and  $p=3$ , parts (a) and (b) illustrate that the result of a multiplication can have either  $2p-1$  or  $2p$  digits; hence, the position where a 1 is added when rounding up (just left of the arrow) can vary. Part (c) shows that rounding up can cause a carry-out.

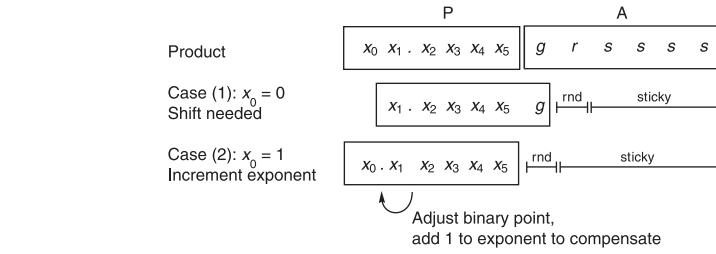
In the figure,  $p=3$ , so the final result must be rounded to three significant digits. The three most-significant digits are in boldface. The fourth most-significant digit (marked with an arrow) is the *round* digit, denoted by  $r$ .

If the round digit is less than 5, then the bold digits represent the rounded result. If the round digit is greater than 5 (as in part (a)), then 1 must be added to the least-significant bold digit. If the round digit is exactly 5 (as in part (b)), then additional digits must be examined to decide between truncation or incrementing by 1. It is only necessary to know if any digits past 5 are nonzero. In the algorithm below, this will be recorded in a *sticky bit*. Comparing parts (a) and (b) in the figure shows that there are two possible positions for the round digit (relative to the least-significant digit of the product). Case (c) illustrates that, when adding 1 to the least-significant bold digit, there may be a carry-out. When this happens, the final significand must be 10.0.

There is a straightforward method of handling rounding using the multiplier of Figure J.2 (page J-4) together with an extra sticky bit. If  $p$  is the number of bits in the significand, then the A, B, and P registers should be  $p$  bits wide. Multiply the two significands to obtain a  $2p$ -bit product in the (P,A) registers (see Figure J.10). During the multiplication, the first  $p-2$  times a bit is shifted into the A register, OR it into the sticky bit. This will be used in halfway cases. Let  $s$  represent the sticky bit,  $g$  (for guard) the most-significant bit of A, and  $r$  (for round) the second most-significant bit of A. There are two cases:

1. The high-order bit of P is 0. Shift P left 1 bit, shifting in the  $g$  bit from A. Shifting the rest of A is not necessary.
2. The high-order bit of P is 1. Set  $s := s \vee r$  and  $r := g$ , and add 1 to the exponent.

Now if  $r=0$ , P is the correctly rounded product. If  $r=1$  and  $s=1$ , then  $P+1$  is the product (where by  $P+1$  we mean adding 1 to the least-significant bit of P).



**Figure J.10** The two cases of the floating-point multiply algorithm. The top line shows the contents of the P and A registers after multiplying the significands, with  $p=6$ . In case (1), the leading bit is 0, and so the P register must be shifted. In case (2), the leading bit is 1, no shift is required, but both the exponent and the round and sticky bits must be adjusted. The sticky bit is the logical OR of the bits marked s.

If  $r=1$  and  $s=0$ , we are in a halfway case and round up according to the least-significant bit of P. As an example, apply the decimal version of these rules to Figure J.9(b). After the multiplication,  $P=126$  and  $A=501$ , with  $g=5$ ,  $r=0$  and  $s=1$ . Since the high-order digit of P is nonzero, case (2) applies and  $r := g$ , so that  $r=5$ , as the arrow indicates in Figure J.9. Since  $r=5$ , we could be in a halfway case, but  $s=1$  indicates that the result is in fact slightly over 1/2, so add 1 to P to obtain the correctly rounded product.

The precise rules for rounding depend on the rounding mode and are given in Figure J.11. Note that P is nonnegative, that is, it contains the magnitude of the result. A good discussion of more efficient ways to implement rounding is in Santoro, Bewick, and Horowitz [1989].

---

**Example** In binary with  $p=4$ , show how the multiplication algorithm computes the product  $-5 \times 10$  in each of the four rounding modes.

**Answer** In binary,  $-5$  is  $-1.010_2 \times 2^2$  and  $10 = 1.010_2 \times 2^3$ . Applying the integer multiplication algorithm to the significands gives  $01100100_2$ , so  $P=0110_2$ ,  $A=0100_2$ ,  $g=0$ ,  $r=1$ , and  $s=0$ . The high-order bit of P is 0, so case (1) applies. Thus, P becomes  $1100_2$ , and since the result is negative, Figure J.11 gives:

|                    |          |                                                                                              |
|--------------------|----------|----------------------------------------------------------------------------------------------|
| round to $-\infty$ | $1101_2$ | add 1 since $r \vee s=1 \wedge 0=\text{TRUE}$                                                |
| round to $+\infty$ | $1100_2$ |                                                                                              |
| round to 0         | $1100_2$ |                                                                                              |
| round to nearest   | $1100_2$ | no add since $r \wedge p_0=1 \wedge 0=\text{FALSE}$ and $r \wedge s=1 \wedge 0=\text{FALSE}$ |

---

The exponent is  $2+3=5$ , so the result is  $-1.100_2 \times 2^5 = -48$ , except when rounding to  $-\infty$ , in which case it is  $-1.101_2 \times 2^5 = -52$ .

| Rounding mode | Sign of result $\geq 0$              | Sign of result $< 0$                 |
|---------------|--------------------------------------|--------------------------------------|
| $-\infty$     |                                      | +1 if $r \vee s$                     |
| $+\infty$     | +1 if $r \vee s$                     |                                      |
| 0             |                                      |                                      |
| Nearest       | +1 if $r \wedge p_0$ or $r \wedge s$ | +1 if $r \wedge p_0$ or $r \wedge s$ |

**Figure J.11 Rules for implementing the IEEE rounding modes.** Let  $S$  be the magnitude of the preliminary result. Blanks mean that the  $p$  most-significant bits of  $S$  are the actual result bits. If the condition listed is true, add 1 to the  $p$ th most-significant bit of  $S$ . The symbols  $r$  and  $s$  represent the round and sticky bits, while  $p_0$  is the  $p$ th most-significant bit of  $S$ .

Overflow occurs when the rounded result is too large to be represented. In single precision, this occurs when the result has an exponent of 128 or higher. If  $e_1$  and  $e_2$  are the two biased exponents, then  $1 \leq e_i \leq 254$ , and the exponent calculation  $e_1 + e_2 - 127$  gives numbers between  $1 + 1 - 127$  and  $254 + 254 - 127$ , or between  $-125$  and  $381$ . This range of numbers can be represented using 9 bits. So one way to detect overflow is to perform the exponent calculations in a 9-bit adder (see Exercise J.12). Remember that you must check for overflow *after* rounding—the example in Figure J.9(c) shows that this can make a difference.

## Denormals

Checking for underflow is somewhat more complex because of denormals. In single precision, if the result has an exponent less than  $-126$ , that does not necessarily indicate underflow, because the result might be a denormal number. For example, the product of  $(1 \times 2^{-64})$  with  $(1 \times 2^{-65})$  is  $1 \times 2^{-129}$ , and  $-129$  is below the legal exponent limit. But this result is a valid denormal number, namely,  $0.125 \times 2^{-126}$ . In general, when the unbiased exponent of a product dips below  $-126$ , the resulting product must be shifted right and the exponent incremented until the exponent reaches  $-126$ . If this process causes the entire significand to be shifted out, then underflow has occurred. The precise definition of underflow is somewhat subtle—see Section J.7 for details.

When one of the operands of a multiplication is denormal, its significand will have leading zeros, and so the product of the significands will also have leading zeros. If the exponent of the product is less than  $-126$ , then the result is denormal, so right-shift and increment the exponent as before. If the exponent is greater than  $-126$ , the result may be a normalized number. In this case, left-shift the product (while decrementing the exponent) until either it becomes normalized or the exponent drops to  $-126$ .

Denormal numbers present a major stumbling block to implementing floating-point multiplication, because they require performing a variable shift in the multiplier, which wouldn't otherwise be needed. Thus, high-performance, floating-point multipliers often do not handle denormalized

numbers, but instead trap, letting software handle them. A few practical codes frequently underflow, even when working properly, and these programs will run quite a bit slower on systems that require denormals to be processed by a trap handler.

So far we haven't mentioned how to deal with operands of zero. This can be handled by either testing both operands before beginning the multiplication or testing the product afterward. If you test afterward, be sure to handle the case  $0 \times \infty$  properly: This results in NaN, not 0. Once you detect that the result is 0, set the biased exponent to 0. Don't forget about the sign. The sign of a product is the XOR of the signs of the operands, even when the result is 0.

### Precision of Multiplication

In the discussion of integer multiplication, we mentioned that designers must decide whether to deliver the low-order word of the product or the entire product. A similar issue arises in floating-point multiplication, where the exact product can be rounded to the precision of the operands or to the next higher precision. In the case of integer multiplication, none of the standard high-level languages contains a construct that would generate a “single times single gets double” instruction. The situation is different for floating point. Many languages allow assigning the product of two single-precision variables to a double-precision one, and the construction can also be exploited by numerical algorithms. The best-known case is using iterative refinement to solve linear systems of equations.

## J.5

### Floating-Point Addition

Typically, a floating-point operation takes two inputs with  $p$  bits of precision and returns a  $p$ -bit result. The ideal algorithm would compute this by first performing the operation exactly, and then rounding the result to  $p$  bits (using the current rounding mode). The multiplication algorithm presented in the previous section follows this strategy. Even though hardware implementing IEEE arithmetic must return the same result as the ideal algorithm, it doesn't need to actually perform the ideal algorithm. For addition, in fact, there are better ways to proceed. To see this, consider some examples.

First, the sum of the binary 6-bit numbers  $1.10011_2$  and  $1.10001_2 \times 2^{-5}$ : When the summands are shifted so they have the same exponent, this is

$$\begin{array}{r} 1.10011 \\ + .0000110001 \\ \hline \end{array}$$

Using a 6-bit adder (and discarding the low-order bits of the second addend) gives

$$\begin{array}{r} 1.10011 \\ + .00001 \\ \hline + 1.10100 \end{array}$$

The first discarded bit is 1. This isn't enough to decide whether to round up. The rest of the discarded bits, 0001, need to be examined. Or, actually, we just need to record whether any of these bits are nonzero, storing this fact in a sticky bit just as in the multiplication algorithm. So, for adding two  $p$ -bit numbers, a  $p$ -bit adder is sufficient, as long as the first discarded bit (round) and the OR of the rest of the bits (sticky) are kept. Then Figure J.11 can be used to determine if a roundup is necessary, just as with multiplication. In the example above, sticky is 1, so a roundup is necessary. The final sum is  $1.10101_2$ .

Here's another example:

$$\begin{array}{r} 1.11011 \\ + .0101001 \\ \hline \end{array}$$

A 6-bit adder gives:

$$\begin{array}{r} 1.11011 \\ + .01010 \\ \hline + 10.00101 \end{array}$$

Because of the carry-out on the left, the round bit isn't the first discarded bit; rather, it is the low-order bit of the sum (1). The discarded bits, 01, are OR'ed together to make sticky. Because round and sticky are both 1, the high-order 6 bits of the sum,  $10.0010_2$ , must be rounded up for the final answer of  $10.0011_2$ .

Next, consider subtraction and the following example:

$$\begin{array}{r} 1.00000 \\ - .00000101111 \\ \hline \end{array}$$

The simplest way of computing this is to convert  $-.00000101111_2$  to its two's complement form, so the difference becomes a sum:

$$\begin{array}{r} 1.00000 \\ + 1.11111010001 \\ \hline \end{array}$$

Computing this sum in a 6-bit adder gives:

$$\begin{array}{r} 1.00000 \\ + 1.11111 \\ \hline 0.11111 \end{array}$$

Because the top bits canceled, the first discarded bit (the guard bit) is needed to fill in the least-significant bit of the sum, which becomes  $0.111110_2$ , and the second discarded bit becomes the round bit. This is analogous to case (1) in the multiplication algorithm (see page J-19). The round bit of 1 isn't enough to decide whether to round up. Instead, we need to OR all the remaining bits (0001) into a sticky bit. In this case, sticky is 1, so the final result must be rounded up to  $0.111111$ . This example shows that if subtraction causes the most-significant bit to cancel, then one guard bit is needed. It is natural to ask whether two guard bits are needed for the case when the *two* most-significant bits cancel. The answer is no, because if  $x$  and  $y$  are so close that the top two bits of  $x - y$  cancel, then  $x - y$  will be exact, so guard bits aren't needed at all.

To summarize, addition is more complex than multiplication because, depending on the signs of the operands, it may actually be a subtraction. If it is an addition, there can be carry-out on the left, as in the second example. If it is subtraction, there can be cancellation, as in the third example. In each case, the position of the round bit is different. However, we don't need to compute the exact sum and then round. We can infer it from the sum of the high-order  $p$  bits together with the round and sticky bits.

The rest of this section is devoted to a detailed discussion of the floating-point addition algorithm. Let  $a_1$  and  $a_2$  be the two numbers to be added. The notations  $e_i$  and  $s_i$  are used for the exponent and significand of the addends  $a_i$ . This means that the floating-point inputs have been unpacked and that  $s_i$  has an explicit leading bit. To add  $a_1$  and  $a_2$ , perform these eight steps:

1. If  $e_1 < e_2$ , swap the operands. This ensures that the difference of the exponents satisfies  $d = e_1 - e_2 \geq 0$ . Tentatively set the exponent of the result to  $e_1$ .
2. If the signs of  $a_1$  and  $a_2$  differ, replace  $s_2$  by its two's complement.
3. Place  $s_2$  in a  $p$ -bit register and shift it  $d = e_1 - e_2$  places to the right (shifting in 1's if  $s_2$  was complemented in the previous step). From the bits shifted out, set  $g$  to the most-significant bit, set  $r$  to the next most-significant bit, and set sticky to the OR of the rest.
4. Compute a preliminary significand  $S = s_1 + s_2$  by adding  $s_1$  to the  $p$ -bit register containing  $s_2$ . If the signs of  $a_1$  and  $a_2$  are different, the most-significant bit of  $S$  is 1, and there was no carry-out, then  $S$  is negative. Replace  $S$  with its two's complement. This can only happen when  $d = 0$ .
5. Shift  $S$  as follows. If the signs of  $a_1$  and  $a_2$  are the same and there was a carryout in step 4, shift  $S$  right by one, filling in the high-order position with 1 (the carry-out). Otherwise, shift it left until it is normalized. When left-shifting, on the first shift fill in the low-order position with the  $g$  bit. After that, shift in zeros. Adjust the exponent of the result accordingly.
6. Adjust  $r$  and  $s$ . If  $S$  was shifted right in step 5, set  $r :=$  low-order bit of  $S$  before shifting and  $s := g$  OR  $r$  OR  $s$ . If there was no shift, set  $r := g$ ,  $s := r$  OR  $s$ . If there was a single left shift, don't change  $r$  and  $s$ . If there were two or more left shifts,  $r := 0$ ,  $s := 0$ . (In the last case, two or more shifts can only happen when  $a_1$  and  $a_2$  have opposite signs and the same exponent, in which case the computation  $s_1 + s_2$  in step 4 will be exact.)
7. Round  $S$  using Figure J.11; namely, if a table entry is nonempty, add 1 to the low-order bit of  $S$ . If rounding causes carry-out, shift  $S$  right and adjust the exponent. This is the significand of the result.
8. Compute the sign of the result. If  $a_1$  and  $a_2$  have the same sign, this is the sign of the result. If  $a_1$  and  $a_2$  have different signs, then the sign of the result depends on which of  $a_1$  or  $a_2$  is negative, whether there was a swap in step 1, and whether  $S$  was replaced by its two's complement in step 4. See Figure J.12.

| swap | compl | sign( $a_1$ ) | sign( $a_2$ ) | sign(result) |
|------|-------|---------------|---------------|--------------|
| Yes  |       | +             | -             | -            |
| Yes  |       | -             | +             | +            |
| No   | No    | +             | -             | +            |
| No   | No    | -             | +             | -            |
| No   | Yes   | +             | -             | -            |
| No   | Yes   | -             | +             | +            |

**Figure J.12 Rules for computing the sign of a sum when the addends have different signs.** The *swap* column refers to swapping the operands in step 1, while the *compl* column refers to performing a two's complement in step 4. Blanks are “don’t care.”

---

**Example** Use the algorithm to compute the sum  $(-1.001_2 \times 2^{-2}) + (-1.111_2 \times 2^0)$ .

**Answer**  $s_1 = 1.001, e_1 = -2, s_2 = 1.111, e_2 = 0$

1.  $e_1 < e_2$ , so swap.  $d = 2$ . Tentative exp = 0.
  2. Signs of both operands negative, don't negate  $s_2$ .
  3. Shift  $s_2$  (1.001 after swap) right by 2, giving  $s_2 = .010, g = 0, r = 1, s = 0$ .
  4. 
$$\begin{array}{r} 1.111 \\ + .010 \\ \hline (1)0.001 \end{array}$$
  $S = 0.001$ , with a carry – out.
  5. Carry-out, so shift  $S$  right,  $S = 1.000$ ,  $\text{exp} = \text{exp} + 1$ , so  $\text{exp} = 1$ .
  6.  $r = \text{low-order bit of sum} = 1, s = g \vee r \vee s = 0 \vee 1 \vee 0 = 1$ .
  7.  $r$  AND  $s = \text{TRUE}$ , so Figure J.11 says round up,  $S = S + 1$  or  $S = 1.001$ .
  8. Both signs negative, so sign of result is negative. Final answer:  $-S \times 2^{\text{exp}} = 1.001_2 \times 2^1$ .
- 

**Example** Use the algorithm to compute the sum  $(-1.010_2) + 1.100_2$ .

**Answer**  $s_1 = 1.010, e_1 = 0, s_2 = 1.100, e_2 = 0$

1. No swap,  $d = 0$ , tentative exp = 0.
2. Signs differ, replace  $s_2$  with 0.100.
3.  $d = 0$ , so no shift.  $r = g = s = 0$ .
4. 
$$\begin{array}{r} 1.010 \\ + 0.100 \\ \hline 1.110 \end{array}$$
 Signs are different, most-significant bit is 1, no carry-out, so must two's complement sum, giving  $S = 0.010$ .

5. Shift left twice, so  $S = 1.000$ ,  $\exp = \exp - 2$ , or  $\exp = -2$ .
  6. Two left shifts, so  $r = g = s = 0$ .
  7. No addition required for rounding.
  8. Answer is sign  $\times S \times 2^{\exp}$  or sign  $\times 1.000 \times 2^{-2}$ . Get sign from Figure J.12. Since complement but no swap and sign( $a_1$ ) is  $-$ , the sign of the sum is  $+$ . Thus, the answer  $= 1.000_2 \times 2^{-2}$ .
- 

## Speeding Up Addition

Let's estimate how long it takes to perform the algorithm above. Step 2 may require an addition, step 4 requires one or two additions, and step 7 may require an addition. If it takes  $T$  time units to perform a  $p$ -bit add (where  $p = 24$  for single precision, 53 for double), then it appears the algorithm will take at least  $4T$  time units. But that is too pessimistic. If step 4 requires two adds, then  $a_1$  and  $a_2$  have the same exponent and different signs, but in that case the difference is exact, so no roundup is required in step 7. Thus, only three additions will ever occur. Similarly, it appears that a variable shift may be required both in step 3 and step 5. But if  $|e_1 - e_2| \leq 1$ , then step 3 requires a right shift of at most one place, so only step 5 needs a variable shift. And, if  $|e_1 - e_2| > 1$ , then step 3 needs a variable shift, but step 5 will require a left shift of at most one place. So only a single variable shift will be performed. Still, the algorithm requires three sequential adds, which, in the case of a 53-bit double-precision significand, can be rather time consuming.

A number of techniques can speed up addition. One is to use pipelining. The "Putting It All Together" section gives examples of how some commercial chips pipeline addition. Another method (used on the Intel 860 [Kohn and Fu 1989]) is to perform two additions in parallel. We now explain how this reduces the latency from  $3T$  to  $T$ .

There are three cases to consider. First, suppose that both operands have the same sign. We want to combine the addition operations from steps 4 and 7. The position of the high-order bit of the sum is not known ahead of time, because the addition in step 4 may or may not cause a carry-out. Both possibilities are accounted for by having two adders. The first adder assumes the add in step 4 will not result in a carry-out. Thus, the values of  $r$  and  $s$  can be computed before the add is actually done. If  $r$  and  $s$  indicate that a roundup is necessary, the first adder will compute  $S = s_1 + s_2 + 1$ , where the notation  $+1$  means adding 1 at the position of the least-significant bit of  $s_1$ . This can be done with a regular adder by setting the low-order carry-in bit to 1. If  $r$  and  $s$  indicate no roundup, the adder computes  $S = s_1 + s_2$  as usual. One extra detail: When  $r = 1$ ,  $s = 0$ , you will also need to know the low-order bit of the sum, which can also be computed in advance very quickly. The second adder covers the possibility that there will be carry-out. The values of  $r$  and  $s$  and the position where the roundup 1 is added are different from above, but again they can be quickly computed in advance. It is not known whether there will be a carry-out until after the add is actually done, but that doesn't matter. By doing both adds in parallel, one adder is guaranteed to reduce the correct answer.

The next case is when  $a_1$  and  $a_2$  have opposite signs but the same exponent. The sum  $a_1 + a_2$  is exact in this case (no roundup is necessary) but the sign isn't known until the add is completed. So don't compute the two's complement (which requires an add) in step 2, but instead compute  $\bar{s}_1 + s_2 + 1$  and  $s_1 + \bar{s}_2 + 1$  in parallel. The first sum has the result of simultaneously complementing  $s_1$  and computing the sum, resulting in  $s_2 - s_1$ . The second sum computes  $s_1 - s_2$ . One of these will be nonnegative and hence the correct final answer. Once again, all the additions are done in one step using two adders operating in parallel.

The last case, when  $a_1$  and  $a_2$  have opposite signs and different exponents, is more complex. If  $|e_1 - e_2| > 1$ , the location of the leading bit of the difference is in one of two locations, so there are two cases just as in addition. When  $|e_1 - e_2| = 1$ , cancellation is possible and the leading bit could be almost anywhere. However, only if the leading bit of the difference is in the same position as the leading bit of  $s_1$  could a roundup be necessary. So one adder assumes a roundup, and the other assumes no roundup. Thus, the addition of step 4 and the rounding of step 7 can be combined. However, there is still the problem of the addition in step 2!

To eliminate this addition, consider the following diagram of step 4:

$$\begin{array}{c} | \quad \quad p \quad \quad | \\ s_1 \quad 1.xxxxxx \\ s_2 - \quad \quad \underline{1yyzzzz} \end{array}$$

If the bits marked  $z$  are all 0, then the high-order  $p$  bits of  $S = s_1 - s_2$  can be computed as  $s_1 + \bar{s}_2 + 1$ . If at least one of the  $z$  bits is 1, use  $s_1 + \bar{s}_2$ . So  $s_1 - s_2$  can be computed with one addition. However, we still don't know  $g$  and  $r$  for the two's complement of  $s_2$ , which are needed for rounding in step 7.

To compute  $s_1 - s_2$  and get the proper  $g$  and  $r$  bits, combine steps 2 and 4 as follows. Don't complement  $s_2$  in step 2. Extend the adder used for computing  $S$  two bits to the right (call the extended sum  $S'$ ). If the preliminary sticky bit (computed in step 3) is 1, compute  $S' = s'_1 + \bar{s}'_2$ , where  $s'_1$  has two 0 bits tacked onto the right, and  $s'_2$  has preliminary  $g$  and  $r$  appended. If the sticky bit is 0, compute  $s'_1 + \bar{s}'_2 + 1$ . Now the two low-order bits of  $S'$  have the correct values of  $g$  and  $r$  (the sticky bit was already computed properly in step 3). Finally, this modification can be combined with the modification that combines the addition from steps 4 and 7 to provide the final result in time  $T$ , the time for one addition.

A few more details need to be considered, as discussed in Santoro, Bewick, and Horowitz [1989] and Exercise J.17. Although the Santoro paper is aimed at multiplication, much of the discussion applies to addition as well. Also relevant is Exercise J.19, which contains an alternative method for adding signed magnitude numbers.

## Denormalized Numbers

Unlike multiplication, for addition very little changes in the preceding description if one of the inputs is a denormal number. There must be a test to see if the exponent field is 0. If it is, then when unpacking the significand there will not be a leading 1.

By setting the biased exponent to 1 when unpacking a denormal, the algorithm works unchanged.

To deal with denormalized outputs, step 5 must be modified slightly. Shift  $S$  until it is normalized, or until the exponent becomes  $E_{\min}$  (that is, the biased exponent becomes 1). If the exponent is  $E_{\min}$  and, after rounding, the high-order bit of  $S$  is 1, then the result is a normalized number and should be packed in the usual way, by omitting the 1. If, on the other hand, the high-order bit is 0, the result is denormal. When the result is unpacked, the exponent field must be set to 0. Section J.7 discusses the exact rules for detecting underflow.

Incidentally, detecting overflow is very easy. It can only happen if step 5 involves a shift right and the biased exponent at that point is bumped up to 255 in single precision (or 2047 for double precision), or if this occurs after rounding.

## J.6

## Division and Remainder

In this section, we'll discuss floating-point division and remainder.

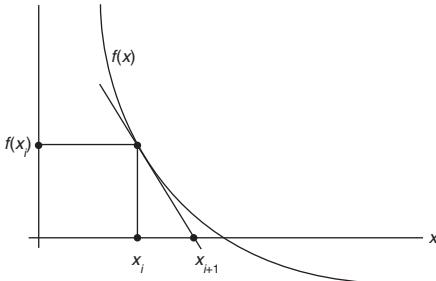
### Iterative Division

We earlier discussed an algorithm for integer division. Converting it into a floating-point division algorithm is similar to converting the integer multiplication algorithm into floating point. The formula

$$(s_1 \times 2^{e_1}) / (s_2 \times 2^{e_2}) = (s_1 / s_2) \times 2^{e_1 - e_2}$$

shows that if the divider computes  $s_1/s_2$ , then the final answer will be this quotient multiplied by  $2^{e_1 - e_2}$ . Referring to Figure J.2(b) (page J-4), the alignment of operands is slightly different from integer division. Load  $s_2$  into B and  $s_1$  into P. The A register is not needed to hold the operands. Then the integer algorithm for division (with the one small change of skipping the very first left shift) can be used, and the result will be of the form  $q_0.q_1\dots$ . To round, simply compute two additional quotient bits (guard and round) and use the remainder as the sticky bit. The guard digit is necessary because the first quotient bit might be 0. However, since the numerator and denominator are both normalized, it is not possible for the two most-significant quotient bits to be 0. This algorithm produces one quotient bit in each step.

A different approach to division converges to the quotient at a quadratic rather than a linear rate. An actual machine that uses this algorithm will be discussed in Section J.10. First, we will describe the two main iterative algorithms, and then we will discuss the pros and cons of iteration when compared with the direct algorithms. A general technique for constructing iterative algorithms, called *Newton's iteration*, is shown in Figure J.13. First, cast the problem in the form of finding the zero of a function. Then, starting from a guess for the zero, approximate the function by its tangent at that guess and form a new guess based



**Figure J.13** Newton's iteration for zero finding. If  $x_i$  is an estimate for a zero of  $f$ , then  $x_{i+1}$  is a better estimate. To compute  $x_{i+1}$ , find the intersection of the  $x$ -axis with the tangent line to  $f$  at  $f(x_i)$ .

on where the tangent has a zero. If  $x_i$  is a guess at a zero, then the tangent line has the equation:

$$y - f(x_i) = f'(x_i)(x - x_i)$$

This equation has a zero at

$$\text{J.6.1} \quad x = x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

To recast division as finding the zero of a function, consider  $f(x) = x^{-1} - b$ . Since the zero of this function is at  $1/b$ , applying Newton's iteration to it will give an iterative method of computing  $1/b$  from  $b$ . Using  $f'(x) = -1/x^2$ , Equation J.6.1 becomes:

$$\text{J.6.2} \quad x_{i+1} = x_i - \frac{1/x_i - b}{-1/x_i^2} = x_i + x_i - x_i^2 b = x_i(2 - x_i b)$$

Thus, we could implement computation of  $a/b$  using the following method:

1. Scale  $b$  to lie in the range  $1 \leq b < 2$  and get an approximate value of  $1/b$  (call it  $x_0$ ) using a table lookup.
2. Iterate  $x_{i+1} = x_i(2 - x_i b)$  until reaching an  $x_n$  that is accurate enough.
3. Compute  $ax_n$  and reverse the scaling done in step 1.

Here are some more details. How many times will step 2 have to be iterated? To say that  $x_i$  is accurate to  $p$  bits means that  $|x_i - 1/b|/(1/b) = 2^{-p}$ , and a simple algebraic manipulation shows that when this is so, then  $(x_{i+1} - 1/b)/(1/b) = 2^{-2p}$ . Thus, the number of correct bits doubles at each step. Newton's iteration is *self-correcting* in the sense that making an error in  $x_i$  doesn't really matter. That is, it treats  $x_i$  as a guess at  $1/b$  and returns  $x_{i+1}$  as an improvement on it (roughly doubling the digits). One thing that would cause  $x_i$  to be in error is rounding error. More

importantly, however, in the early iterations we can take advantage of the fact that we don't expect many correct bits by performing the multiplication in reduced precision, thus gaining speed without sacrificing accuracy. Another application of Newton's iteration is discussed in Exercise J.20.

The second iterative division method is sometimes called *Goldschmidt's algorithm*. It is based on the idea that to compute  $a/b$ , you should multiply the numerator and denominator by a number  $r$  with  $rb \approx 1$ . In more detail, let  $x_0 = a$  and  $y_0 = b$ . At each step compute  $x_{i+1} = r_i x_i$  and  $y_{i+1} = r_i y_i$ . Then the quotient  $x_{i+1}/y_{i+1} = x_i/y_i = a/b$  is constant. If we pick  $r_i$  so that  $y_i \rightarrow 1$ , then  $x_i \rightarrow a/b$ , so the  $x_i$  converge to the answer we want. This same idea can be used to compute other functions. For example, to compute the square root of  $a$ , let  $x_0 = a$  and  $y_0 = a$ , and at each step compute  $x_{i+1} = r_i^2 x_i$ ,  $y_{i+1} = r_i y_i$ . Then  $x_{i+1}/y_{i+1}^2 = x_i/y_i^2 = 1/a$ , so if the  $r_i$  are chosen to drive  $x_i \rightarrow 1$ , then  $y_i \rightarrow \sqrt{a}$ . This technique is used to compute square roots on the TI 8847.

Returning to Goldschmidt's division algorithm, set  $x_0 = a$  and  $y_0 = b$ , and write  $b = 1 - \delta$ , where  $|\delta| < 1$ . If we pick  $r_0 = 1 + \delta$ , then  $y_1 = r_0 y_0 = 1 - \delta^2$ . We next pick  $r_1 = 1 + \delta^2$ , so that  $y_2 = r_1 y_1 = 1 - \delta^4$ , and so on. Since  $|\delta| < 1$ ,  $y_i \rightarrow 1$ . With this choice of  $r_i$ , the  $x_i$  will be computed as  $x_{i+1} = r_i x_i = (1 + \delta^{2^i}) x_i = (1 + (1 - b)^{2^i}) x_i$ , or

$$\text{J.6.3} \quad x_{i+1} = a [1 + (1 - b)] [1 + (1 - b)^2] [1 + (1 - b)^4] \cdots [1 + (1 - b)^{2^i}]$$

There appear to be two problems with this algorithm. First, convergence is slow when  $b$  is not near 1 (that is,  $\delta$  is not near 0), and, second, the formula isn't self-correcting—since the quotient is being computed as a product of independent terms, an error in one of them won't get corrected. To deal with slow convergence, if you want to compute  $a/b$ , look up an approximate inverse to  $b$  (call it  $b'$ ), and run the algorithm on  $ab'/bb'$ . This will converge rapidly since  $bb' \approx 1$ .

To deal with the self-correction problem, the computation should be run with a few bits of extra precision to compensate for rounding errors. However, Goldschmidt's algorithm does have a weak form of self-correction, in that the precise value of the  $r_i$  does not matter. Thus, in the first few iterations, when the full precision of  $1 - \delta^{2^i}$  is not needed you can choose  $r_i$  to be a truncation of  $1 + \delta^{2^i}$ , which may make these iterations run faster without affecting the speed of convergence. If  $r_i$  is truncated, then  $y_i$  is no longer exactly  $1 - \delta^{2^i}$ . Thus, Equation J.6.3 can no longer be used, but it is easy to organize the computation so that it does not depend on the precise value of  $r_i$ . With these changes, Goldschmidt's algorithm is as follows (the notes in brackets show the connection with our earlier formulas).

1. Scale  $a$  and  $b$  so that  $1 \leq b < 2$ .
2. Look up an approximation to  $1/b$  (call it  $b'$ ) in a table.
3. Set  $x_0 = ab'$  and  $y_0 = bb'$ .

4. Iterate until  $x_i$  is close enough to  $a/b$ :

Loop

$$r \approx 2 - y \quad [\text{if } y_i = 1 + \delta_i, \text{ then } r \approx 1 - \delta_i]$$

$$y = y \times r \quad [y_{i+1} = y_i \times r \approx 1 - \delta_i^2]$$

$$x_{i+1} = x_i \times r \quad [x_{i+1} = x_i \times r]$$

End loop

The two iteration methods are related. Suppose in Newton's method that we unroll the iteration and compute each term  $x_{i+1}$  directly in terms of  $b$ , instead of recursively in terms of  $x_i$ . By carrying out this calculation (see Exercise J.22), we discover that

$$x_{i+1} = x_0(2 - x_0b) \left[ \left( 1 + (x_0b - 1)^2 \right) \left( 1 + (x_0b - 1)^4 \right) \cdots \left( 1 + (x_0b - 1)^{2^i} \right) \right]$$

This formula is very similar to Equation J.6.3. In fact, they are identical if  $a$  and  $b$  in J.6.3 are replaced with  $ax_0$ ,  $bx_0$ , and  $a = 1$ . Thus, if the iterations were done to infinite precision, the two methods would yield exactly the same sequence  $x_i$ .

The advantage of iteration is that it doesn't require special divide hardware. Instead, it can use the multiplier (which, however, requires extra control). Further, on each step, it delivers twice as many digits as in the previous step—unlike ordinary division, which produces a fixed number of digits at every step.

There are two disadvantages with inverting by iteration. The first is that the IEEE standard requires division to be correctly rounded, but iteration only delivers a result that is close to the correctly rounded answer. In the case of Newton's iteration, which computes  $1/b$  instead of  $a/b$  directly, there is an additional problem. Even if  $1/b$  were correctly rounded, there is no guarantee that  $a/b$  will be. An example in decimal with  $p = 2$  is  $a = 13$ ,  $b = 51$ . Then  $a/b = .2549\dots$ , which rounds to  $.25$ . But  $1/b = .0196\dots$ , which rounds to  $.020$ , and then  $a \times .020 = .26$ , which is off by 1. The second disadvantage is that iteration does not give a remainder. This is especially troublesome if the floating-point divide hardware is being used to perform integer division, since a remainder operation is present in almost every high-level language.

Traditional folklore has held that the way to get a correctly rounded result from iteration is to compute  $1/b$  to slightly more than  $2p$  bits, compute  $a/b$  to slightly more than  $2p$  bits, and then round to  $p$  bits. However, there is a faster way, which apparently was first implemented on the TI 8847. In this method,  $a/b$  is computed to about 6 extra bits of precision, giving a preliminary quotient  $q$ . By comparing  $qb$  with  $a$  (again with only 6 extra bits), it is possible to quickly decide whether  $q$  is correctly rounded or whether it needs to be bumped up or down by 1 in the least-significant place. This algorithm is explored further in Exercise J.21.

One factor to take into account when deciding on division algorithms is the relative speed of division and multiplication. Since division is more complex than multiplication, it will run more slowly. A common rule of thumb is that division algorithms should try to achieve a speed that is about one-third that of multiplication.

One argument in favor of this rule is that there are real programs (such as some versions of spice) where the ratio of division to multiplication is 1:3. Another place where a factor of 3 arises is in the standard iterative method for computing square root. This method involves one division per iteration, but it can be replaced by one using three multiplications. This is discussed in Exercise J.20.

## Floating-Point Remainder

For nonnegative integers, integer division and remainder satisfy:

$$a = (a \text{ DIV } b)b + a \text{ REM } b, \quad 0 \leq a \text{ REM } b < b$$

A floating-point remainder  $x \text{ REM } y$  can be similarly defined as  $x = \text{INT}(x/y)y + x \text{ REM } y$ . How should  $x/y$  be converted to an integer? The IEEE remainder function uses the round-to-even rule. That is, pick  $n = \text{INT}(x/y)$  so that  $|x/y - n| \leq 1/2$ . If two different  $n$  satisfy this relation, pick the even one. Then  $\text{REM}$  is defined to be  $x - yn$ . Unlike integers where  $0 \leq a \text{ REM } b < b$ , for floating-point numbers  $|x \text{ REM } y| \leq y/2$ . Although this defines  $\text{REM}$  precisely, it is not a practical operational definition, because  $n$  can be huge. In single precision,  $n$  could be as large as  $2^{127}/2^{-126} = 2^{253} \approx 10^{76}$ .

There is a natural way to compute  $\text{REM}$  if a direct division algorithm is used. Proceed as if you were computing  $x/y$ . If  $x = s_1 2^{e_1}$  and  $y = s_2 2^{e_2}$  and the divider is as in Figure J.2(b) (page J-4), then load  $s_1$  into P and  $s_2$  into B. After  $e_1 - e_2$  division steps, the P register will hold a number  $r$  of the form  $x - yn$  satisfying  $0 \leq r < y$ . Since the IEEE remainder satisfies  $|\text{REM}| \leq y/2$ ,  $\text{REM}$  is equal to either  $r$  or  $r - y$ . It is only necessary to keep track of the last quotient bit produced, which is needed to resolve halfway cases. Unfortunately,  $e_1 - e_2$  can be a lot of steps, and floating-point units typically have a maximum amount of time they are allowed to spend on one instruction. Thus, it is usually not possible to implement  $\text{REM}$  directly. None of the chips discussed in Section J.10 implements  $\text{REM}$ , but they could by providing a remainder-step instruction—this is what is done on the Intel 8087 family. A remainder step takes as arguments two numbers  $x$  and  $y$ , and performs divide steps until either the remainder is in P or  $n$  steps have been performed, where  $n$  is a small number, such as the number of steps required for division in the highest-supported precision. Then  $\text{REM}$  can be implemented as a software routine that calls the  $\text{REM}$  step instruction  $\lfloor (e_1 - e_2)/n \rfloor$  times, initially using  $x$  as the numerator but then replacing it with the remainder from the previous  $\text{REM}$  step.

$\text{REM}$  can be used for computing trigonometric functions. To simplify things, imagine that we are working in base 10 with five significant figures, and consider computing  $\sin x$ . Suppose that  $x = 7$ . Then we can reduce by  $\pi = 3.1416$  and compute  $\sin(7) = \sin(7 - 2 \times 3.1416) = \sin(0.7168)$  instead. But, suppose we want to compute  $\sin(2.0 \times 10^5)$ . Then  $2 \times 10^5 / 3.1416 = 63661.8$ , which in our five-place system comes out to be 63662. Since multiplying 3.1416 times 63662 gives 200000.5392, which rounds to  $2.0000 \times 10^5$ , argument reduction reduces  $2 \times 10^5$  to 0, which is not even close to being correct. The problem is that our

five-place system does not have the precision to do correct argument reduction. Suppose we had the REM operator. Then we could compute  $2 \times 10^5 \text{ REM } 3.1416$  and get  $-.53920$ . However, this is still not correct because we used 3.1416, which is an approximation for  $\pi$ . The value of  $2 \times 10^5 \text{ REM } \pi$  is  $-.071513$ .

Traditionally, there have been two approaches to computing periodic functions with large arguments. The first is to return an error for their value when  $x$  is large. The second is to store  $\pi$  to a very large number of places and do exact argument reduction. The REM operator is not much help in either of these situations. There is a third approach that has been used in some math libraries, such as the Berkeley UNIX 4.3bsd release. In these libraries,  $\pi$  is computed to the nearest floating-point number. Let's call this machine  $\pi$ , and denote it by  $\pi'$ . Then, when computing  $\sin x$ , reduce  $x$  using  $x \text{ REM } \pi'$ . As we saw in the above example,  $x \text{ REM } \pi'$  is quite different from  $x \text{ REM } \pi$  when  $x$  is large, so that computing  $\sin x$  as  $\sin(x \text{ REM } \pi')$  will not give the exact value of  $\sin x$ . However, computing trigonometric functions in this fashion has the property that all familiar identities (such as  $\sin^2 x + \cos^2 x = 1$ ) are true to within a few rounding errors. Thus, using REM together with machine  $\pi$  provides a simple method of computing trigonometric functions that is accurate for small arguments and still may be useful for large arguments.

When REM is used for argument reduction, it is very handy if it also returns the low-order bits of  $n$  (where  $x \text{ REM } y = x - ny$ ). This is because a practical implementation of trigonometric functions will reduce by something smaller than  $2\pi$ . For example, it might use  $\pi/2$ , exploiting identities such as  $\sin(x - \pi/2) = -\cos x$ ,  $\sin(x - \pi) = -\sin x$ . Then the low bits of  $n$  are needed to choose the correct identity.

## J.7

## More on Floating-Point Arithmetic

Before leaving the subject of floating-point arithmetic, we present a few additional topics.

### Fused Multiply-Add

Probably the most common use of floating-point units is performing matrix operations, and the most frequent matrix operation is multiplying a matrix times a matrix (or vector), which boils down to computing an inner product,  $x_1 \cdot y_1 + x_2 \cdot y_2 + \dots + x_n \cdot y_n$ . Computing this requires a series of multiply-add combinations.

Motivated by this, the IBM RS/6000 introduced a single instruction that computes  $ab + c$ , the *fused multiply-add*. Although this requires being able to read three operands in a single instruction, it has the potential for improving the performance of computing inner products.

The fused multiply-add computes  $ab + c$  exactly and then rounds. Although rounding only once increases the accuracy of inner products somewhat, that is not its primary motivation. There are two main advantages of rounding once. First,

as we saw in the previous sections, rounding is expensive to implement because it may require an addition. By rounding only once, an addition operation has been eliminated. Second, the extra accuracy of fused multiply-add can be used to compute correctly rounded division and square root when these are not available directly in hardware. Fused multiply-add can also be used to implement efficient floating-point multiple-precision packages.

The implementation of correctly rounded division using fused multiply-add has many details, but the main idea is simple. Consider again the example from Section J.6 (page J-30), which was computing  $a/b$  with  $a=13$ ,  $b=51$ . Then  $1/b$  rounds to  $b'=.020$ , and  $ab'$  rounds to  $q'=.26$ , which is not the correctly rounded quotient. Applying fused multiply-add twice will correctly adjust the result, via the formulas

$$\begin{aligned} r &= a - bq' \\ q'' &= q' + rb' \end{aligned}$$

Computing to two-digit accuracy,  $bq'=51 \times .26$  rounds to 13, and so  $r=a-bq'$  would be 0, giving no adjustment. But using fused multiply-add gives  $r=a-bq'=13-(51 \times .26)=-.26$ , and then  $q''=q'+rb'=.26-.0052=.2548$ , which rounds to the correct quotient, .25. More details can be found in the papers by Montoye, Hokenek, and Runyon [1990] and Markstein [1990].

## Precisions

The standard specifies four precisions: *single*, *single extended*, *double*, and *double extended*. The properties of these precisions are summarized in Figure J.7 (page J-16). Implementations are not required to have all four precisions, but are encouraged to support either the combination of single and single extended or all of single, double, and double extended. Because of the widespread use of double precision in scientific computing, double precision is almost always implemented. Thus, the computer designer usually only has to decide whether to support double extended and, if so, how many bits it should have.

The Motorola 68882 and Intel 387 coprocessors implement extended precision using the smallest allowable size of 80 bits (64 bits of significand). However, many of the more recently designed, high-performance floating-point chips do not implement 80-bit extended precision. One reason is that the 80-bit width of extended precision is awkward for 64-bit buses and registers. Some new architectures, such as SPARC V8 and PA-RISC, specify a 128-bit extended (or *quad*) precision. They have established a *de facto* convention for quad that has 15 bits of exponent and 113 bits of significand.

Although most high-level languages do not provide access to extended precision, it is very useful to writers of mathematical software. As an example, consider writing a library routine to compute the length of a vector  $(x,y)$  in the plane, namely,  $\sqrt{x^2+y^2}$ . If  $x$  is larger than  $2^{E_{\max}/2}$ , then computing this in the obvious way will overflow. This means that either the allowable exponent range for this subroutine

will be cut in half or a more complex algorithm using scaling will have to be employed. But, if extended precision is available, then the simple algorithm will work. Computing the length of a vector is a simple task, and it is not difficult to come up with an algorithm that doesn't overflow. However, there are more complex problems for which extended precision means the difference between a simple, fast algorithm and a much more complex one. One of the best examples of this is binary-to-decimal conversion. An efficient algorithm for binary-to-decimal conversion that makes essential use of extended precision is very readably presented in Coonen [1984]. This algorithm is also briefly sketched in Goldberg [1991]. Computing accurate values for transcendental functions is another example of a problem that is made much easier if extended precision is present.

One very important fact about precision concerns *double rounding*. To illustrate in decimals, suppose that we want to compute  $1.9 \times 0.66$  and that single precision is two digits, while extended precision is three digits. The exact result of the product is 1.254. Rounded to extended precision, the result is 1.25. When further rounded to single precision, we get 1.2. However, the result of  $1.9 \times 0.66$  correctly rounded to single precision is 1.3. Thus, rounding twice may not produce the same result as rounding once. Suppose you want to build hardware that only does double-precision arithmetic. Can you simulate single precision by computing first in double precision and then rounding to single? The above example suggests that you can't. However, double rounding is not always dangerous. In fact, the following rule is true (this is not easy to prove, but see Exercise J.25).

*If  $x$  and  $y$  have  $p$ -bit significands, and  $x+y$  is computed exactly and then rounded to  $q$  places, a second rounding to  $p$  places will not change the answer if  $q \geq 2p+2$ . This is true not only for addition, but also for multiplication, division, and square root.*

In our example above,  $q=3$  and  $p=2$ , so  $q \geq 2p+2$  is not true. On the other hand, for IEEE arithmetic, double precision has  $q=53$  and  $p=24$ , so  $q=53 \geq 2p+2=50$ . Thus, single precision can be implemented by computing in double precision—that is, computing the answer exactly and then rounding to double—and then rounding to single precision.

## Exceptions

The IEEE standard defines five exceptions: underflow, overflow, divide by zero, inexact, and invalid. By default, when these exceptions occur, they merely set a flag and the computation continues. The flags are *sticky*, meaning that once set they remain set until explicitly cleared. The standard strongly encourages implementations to provide a trap-enable bit for each exception. When an exception with an enabled trap handler occurs, a user trap handler is called, and the value of the associated exception flag is undefined. In Section J.3 we mentioned that  $\sqrt{-3}$  has the value NaN and  $1/0$  is  $\infty$ . These are examples of operations that raise an exception.

By default, computing  $\sqrt{-3}$  sets the invalid flag and returns the value NaN. Similarly 1/0 sets the divide-by-zero flag and returns  $\infty$ .

The underflow, overflow, and divide-by-zero exceptions are found in most other systems. The *invalid exception* is for the result of operations such as  $\sqrt{-1}$ , 0/0, or  $\infty - \infty$ , which don't have any natural value as a floating-point number or as  $\pm\infty$ . The *inexact exception* is peculiar to IEEE arithmetic and occurs either when the result of an operation must be rounded or when it overflows. In fact, since 1/0 and an operation that overflows both deliver  $\infty$ , the exception flags must be consulted to distinguish between them. The inexact exception is an unusual "exception," in that it is not really an exceptional condition because it occurs so frequently. Thus, enabling a trap handler for the inexact exception will most likely have a severe impact on performance. Enabling a trap handler doesn't affect whether an operation is exceptional except in the case of underflow. This is discussed below.

The IEEE standard assumes that when a trap occurs, it is possible to identify the operation that trapped and its operands. On machines with pipelining or multiple arithmetic units, when an exception occurs, it may not be enough to simply have the trap handler examine the program counter. Hardware support may be necessary to identify exactly which operation trapped.

Another problem is illustrated by the following program fragment.

```
r1 = r2/r3
r2 = r4 + r5
```

These two instructions might well be executed in parallel. If the divide traps, its argument  $r_2$  could already have been overwritten by the addition, especially since addition is almost always faster than division. Computer systems that support trapping in the IEEE standard must provide some way to save the value of  $r_2$ , either in hardware or by having the compiler avoid such a situation in the first place. This kind of problem is not peculiar to floating point. In the sequence

```
r1 = 0(r2)
r2 = r3
```

it would be efficient to execute  $r_2 = r_3$  while waiting for memory. But, if accessing  $0(r_2)$  causes a page fault,  $r_2$  might no longer be available for restarting the instruction  $r_1 = 0(r_2)$ .

One approach to this problem, used in the MIPS R3010, is to identify instructions that may cause an exception early in the instruction cycle. For example, an addition can overflow only if one of the operands has an exponent of  $E_{\max}$ , and so on. This early check is conservative: It might flag an operation that doesn't actually cause an exception. However, if such false positives are rare, then this technique will have excellent performance. When an instruction is tagged as being possibly exceptional, special code in a trap handler can compute it without destroying any state. Remember that all these problems occur only when trap handlers are enabled. Otherwise, setting the exception flags during normal processing is straightforward.

## Underflow

We have alluded several times to the fact that detection of underflow is more complex than for the other exceptions. The IEEE standard specifies that if an underflow trap handler is enabled, the system must trap if the result is denormal. On the other hand, if trap handlers are disabled, then the underflow flag is set only if there is a loss of accuracy—that is, if the result must be rounded. The rationale is, if no accuracy is lost on an underflow, there is no point in setting a warning flag. But if a trap handler is enabled, the user might be trying to simulate flush-to-zero and should therefore be notified whenever a result dips below  $1.0 \times 2^{E_{\min}}$ .

So if there is no trap handler, the underflow exception is signaled only when the result is denormal and inexact, but the definitions of *denormal* and *inexact* are both subject to multiple interpretations. Normally, *inexact* means there was a result that couldn't be represented exactly and had to be rounded. Consider the example (in a base 2 floating-point system with 3-bit significands) of  $(1.11_2 \times 2^{-2}) \times (1.11_2 \times 2^{E_{\min}}) = 0.110001_2 \times 2^{E_{\min}}$ , with round to nearest in effect. The delivered result is  $0.11_2 \times 2^{E_{\min}}$ , which had to be rounded, causing *inexact* to be signaled. But is it correct to also signal underflow? Gradual underflow loses significance because the exponent range is bounded. If the exponent range were unbounded, the delivered result would be  $1.10_2 \times 2^{E_{\min}-1}$ , exactly the same answer obtained with gradual underflow. The fact that denormalized numbers have fewer bits in their significand than normalized numbers therefore doesn't make any difference in this case. The commentary to the standard [Cody et al. 1984] encourages this as the criterion for setting the underflow flag. That is, it should be set whenever the delivered result is different from what would be delivered in a system with the same fraction size, but with a very large exponent range. However, owing to the difficulty of implementing this scheme, the standard allows setting the underflow flag whenever the result is denormal and different from the infinitely precise result.

There are two possible definitions of what it means for a result to be denormal. Consider the example of  $1.10_2 \times 2^{-1}$  multiplied by  $1.10_2 \times 2^{E_{\min}}$ . The exact product is  $0.1111 \times 2^{E_{\min}}$ . The rounded result is the normal number  $1.00_2 \times 2^{E_{\min}}$ . Should underflow be signaled? Signaling underflow means that you are using the *before rounding* rule, because the result was denormal before rounding. Not signaling underflow means that you are using the *after rounding* rule, because the result is normalized after rounding. The IEEE standard provides for choosing either rule; however, the one chosen must be used consistently for all operations.

To illustrate these rules, consider floating-point addition. When the result of an addition (or subtraction) is denormal, it is always exact. Thus, the underflow flag never needs to be set for addition. That's because if traps are not enabled then no exception is raised. And if traps are enabled, the value of the underflow flag is undefined, so again it doesn't need to be set.

One final subtlety should be mentioned concerning underflow. When there is no underflow trap handler, the result of an operation on  $p$ -bit numbers that causes

an underflow is a denormal number with  $p - 1$  or fewer bits of precision. When traps are enabled, the trap handler is provided with the result of the operation rounded to  $p$  bits and with the exponent wrapped around. Now there is a potential double-rounding problem. If the trap handler wants to return the denormal result, it can't just round its argument, because that might lead to a double-rounding error. Thus, the trap handler must be passed at least one extra bit of information if it is to be able to deliver the correctly rounded result.

## J.8

## Speeding Up Integer Addition

The previous section showed that many steps go into implementing floating-point operations; however, each floating-point operation eventually reduces to an integer operation. Thus, increasing the speed of integer operations will also lead to faster floating point.

Integer addition is the simplest operation and the most important. Even for programs that don't do explicit arithmetic, addition must be performed to increment the program counter and to calculate addresses. Despite the simplicity of addition, there isn't a single best way to perform high-speed addition. We will discuss three techniques that are in current use: carry-lookahead, carry-skip, and carry-select.

### Carry-Lookahead

An  $n$ -bit adder is just a combinational circuit. It can therefore be written by a logic formula whose form is a sum of products and can be computed by a circuit with two levels of logic. How do you figure out what this circuit looks like? From Equation J.2.1 (page J-3) the formula for the  $i$ th sum can be written as:

J.8.1

$$s_i = a_i \bar{b}_i \bar{c}_i + \bar{a}_i b_i \bar{c}_i + \bar{a}_i \bar{b}_i c_i + a_i b_i c_i$$

where  $c_i$  is both the carry-in to the  $i$ th adder and the carry-out from the  $(i - 1)$ -st adder.

The problem with this formula is that, although we know the values of  $a_i$  and  $b_i$ —they are inputs to the circuit—we don't know  $c_i$ . So our goal is to write  $c_i$  in terms of  $a_i$  and  $b_i$ . To accomplish this, we first rewrite Equation J.2.2 (page J-3) as:

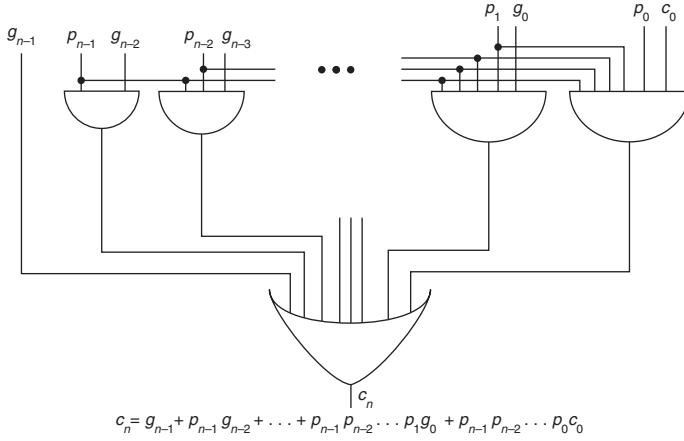
J.8.2

$$c_i = g_{i-1} + p_{i-1} c_{i-1}, \quad g_{i-1} = a_{i-1} b_{i-1}, \quad p_{i-1} = a_{i-1} + b_{i-1}$$

Here is the reason for the symbols  $p$  and  $g$ : If  $g_{i-1}$  is true, then  $c_i$  is certainly true, so a carry is *generated*. Thus,  $g$  is for generate. If  $p_{i-1}$  is true, then if  $c_{i-1}$  is true, it is *propagated* to  $c_i$ . Start with Equation J.8.1 and use Equation J.8.2 to replace  $c_i$  with  $g_{i-1} + p_{i-1} c_{i-1}$ . Then, use Equation J.8.2 with  $i - 1$  in place of  $i$  to replace  $c_{i-1}$  with  $c_{i-2}$ , and so on. This gives the result:

J.8.3

$$c_i = g_{i-1} + p_{i-1} g_{i-2} + p_{i-1} p_{i-2} g_{i-3} + \cdots + p_{i-1} p_{i-2} \cdots p_1 g_0 + p_{i-1} p_{i-2} \cdots p_1 p_0 c_0$$



**Figure J.14** Pure carry-lookahead circuit for computing the carry-out  $c_n$  of an  $n$ -bit adder.

An adder that computes carries using Equation J.8.3 is called a *carry-lookahead adder*, or CLA. A CLA requires one logic level to form  $p$  and  $g$ , two levels to form the carries, and two for the sum, for a grand total of five logic levels. This is a vast improvement over the  $2n$  levels required for the ripple-carry adder.

Unfortunately, as is evident from Equation J.8.3 or from Figure J.14, a carry-lookahead adder on  $n$  bits requires a fan-in of  $n+1$  at the OR gate as well as at the rightmost AND gate. Also, the  $p_{n-1}$  signal must drive  $n$  AND gates. In addition, the rather irregular structure and many long wires of Figure J.14 make it impractical to build a full carry-lookahead adder when  $n$  is large.

However, we can use the carry-lookahead idea to build an adder that has about  $\log_2 n$  logic levels (substantially fewer than the  $2n$  required by a ripplecarry adder) and yet has a simple, regular structure. The idea is to build up the  $p$ 's and  $g$ 's in steps. We have already seen that

$$c_1 = g_0 + c_0 p_0$$

This says there is a carry-out of the 0th position ( $c_1$ ) either if there is a carry generated in the 0th position or if there is a carry into the 0th position and the carry propagates. Similarly,

$$c_2 = G_{01} + P_{01}c_0$$

$G_{01}$  means there is a carry generated out of the block consisting of the first two bits.  $P_{01}$  means that a carry propagates through this block.  $P$  and  $G$  have the following logic equations:

$$\begin{aligned} G_{01} &= g_1 + p_1 g_0 \\ P_{01} &= p_1 p_0 \end{aligned}$$

More generally, for any  $j$  with  $i < j, j+1 < k$ , we have the recursive relations:

$$J.8.4 \quad c_{k+1} = G_{ik} + P_{ik}c_i$$

$$J.8.5 \quad G_{ik} = G_{j+1,k} + P_{j+1,k}G_{ij}$$

$$J.8.6 \quad P_{ik} = P_{ij}P_{j+1,k}$$

Equation J.8.5 says that a carry is generated out of the block consisting of bits  $i$  through  $k$  inclusive if it is generated in the high-order part of the block ( $j+1, k$ ) or if it is generated in the low-order part of the block ( $i, j$ ) and then propagated through the high part. These equations will also hold for  $i \leq j < k$  if we set  $G_{ii} = g_i$  and  $P_{ii} = p_i$ .

**Example** Express  $P_{03}$  and  $G_{03}$  in terms of  $p$ 's and  $g$ 's.

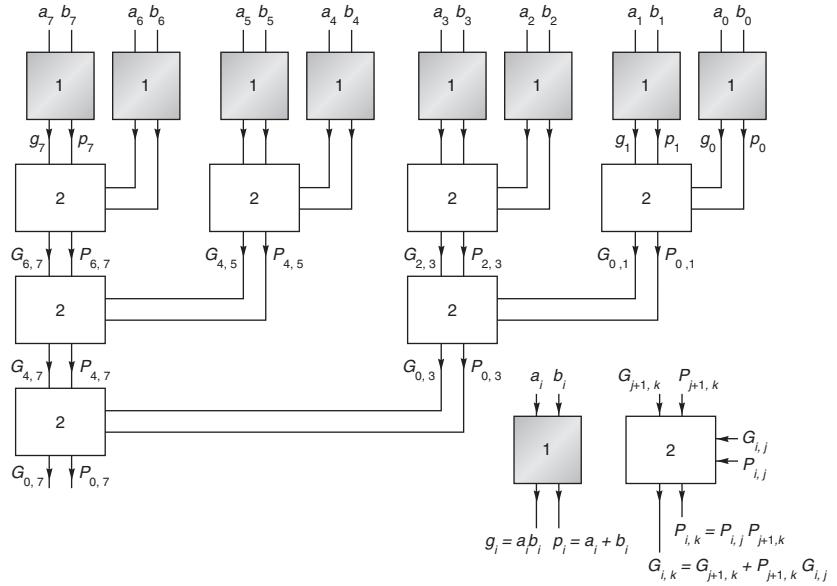
**Answer** Using Equation J.8.6,  $P_{03} = P_{01}P_{23} = P_{00}P_{11}P_{22}P_{33}$ . Since  $P_{ii} = p_i$ ,  $P_{03} = p_0p_1p_2p_3$ . For  $G_{03}$ , Equation J.8.5 says  $G_{03} = G_{23} + P_{23}G_{01} = (G_{33} + P_{33}G_{22}) + (P_{22}P_{33})(G_{11} + P_{11}G_{00}) = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0$ .

---

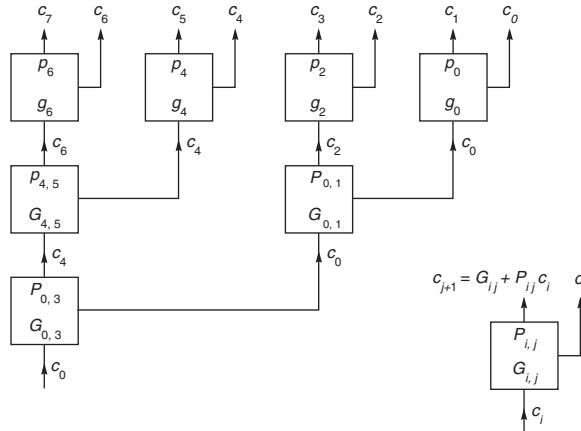
With these preliminaries out of the way, we can now show the design of a practical CLA. The adder consists of two parts. The first part computes various values of  $P$  and  $G$  from  $p_i$  and  $g_i$ , using Equations J.8.5 and J.8.6; the second part uses these  $P$  and  $G$  values to compute all the carries via Equation J.8.4. The first part of the design is shown in Figure J.15. At the top of the diagram, input numbers  $a_7 \dots a_0$  and  $b_7 \dots b_0$  are converted to  $p$ 's and  $g$ 's using cells of type 1. Then various  $P$ 's and  $G$ 's are generated by combining cells of type 2 in a binary tree structure. The second part of the design is shown in Figure J.16. By feeding  $c_0$  in at the bottom of this tree, all the carry bits come out at the top. Each cell must know a pair of  $(P, G)$  values in order to do the conversion, and the value it needs is written inside the cells. Now compare Figures J.15 and J.16. There is a one-to-one correspondence between cells, and the value of  $(P, G)$  needed by the carry-generating cells is exactly the value known by the corresponding  $(P, G)$ -generating cells. The combined cell is shown in Figure J.17. The numbers to be added flow into the top and downward through the tree, combining with  $c_0$  at the bottom and flowing back up the tree to form the carries. Note that one thing is missing from Figure J.17: a small piece of extra logic to compute  $c_8$  for the carry-out of the adder.

The bits in a CLA must pass through about  $\log_2 n$  logic levels, compared with  $2n$  for a ripple-carry adder. This is a substantial speed improvement, especially for a large  $n$ . Whereas the ripple-carry adder had  $n$  cells, however, the CLA has  $2n$  cells, although in our layout they will take  $n \log n$  space. The point is that a small investment in size pays off in a dramatic improvement in speed.

A number of technology-dependent modifications can improve CLAs. For example, if each node of the tree has three inputs instead of two, then the height

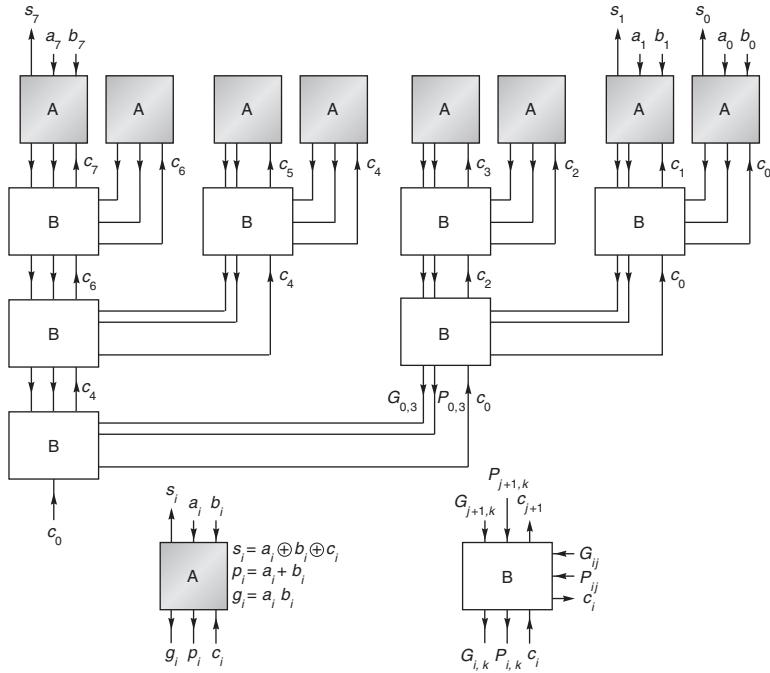


**Figure J.15** First part of carry-lookahead tree. As signals flow from the top to the bottom, various values of  $P$  and  $G$  are computed.



**Figure J.16** Second part of carry-lookahead tree. Signals flow from the bottom to the top, combining with  $P$  and  $G$  to form the carries.

of the tree will decrease from  $\log_2 n$  to  $\log_3 n$ . Of course, the cells will be more complex and thus might operate more slowly, negating the advantage of the decreased height. For technologies where rippling works well, a hybrid design might be better. This is illustrated in Figure J.19. Carries ripple between adders



**Figure J.17 Complete carry-lookahead tree adder.** This is the combination of Figures J.15 and J.16. The numbers to be added enter at the top, flow to the bottom to combine with  $c_0$ , and then flow back up to compute the sum bits.

at the top level, while the “B” boxes are the same as those in Figure J.17. This design will be faster if the time to ripple between four adders is faster than the time it takes to traverse a level of “B” boxes. (To make the pattern more clear, Figure J.19 shows a 16-bit adder, so the 8-bit adder of Figure J.17 corresponds to the right half of Figure J.19.)

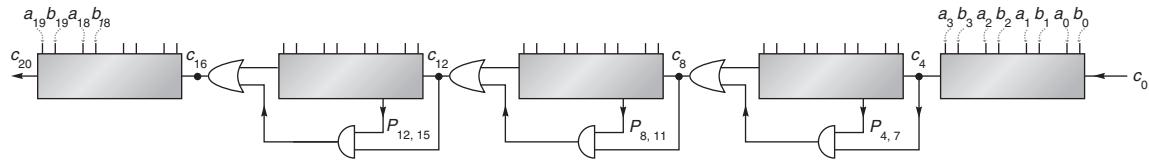
### Carry-Skip Adders

A *carry-skip adder* sits midway between a ripple-carry adder and a carry-lookahead adder, both in terms of speed and cost. (A carry-skip adder is not called a CSA, as that name is reserved for carry-save adders.) The motivation for this adder comes from examining the equations for  $P$  and  $G$ . For example,

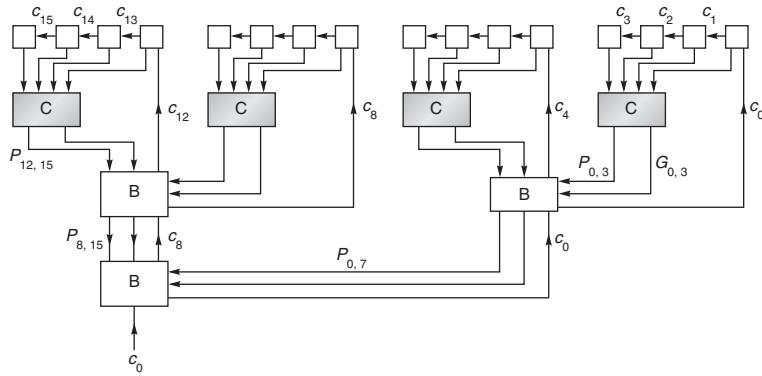
$$P_{03} = p_0 p_1 p_2 p_3$$

$$G_{03} = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$$

Computing  $P$  is much simpler than computing  $G$ , and a carry-skip adder only computes the  $P$ 's. Such an adder is illustrated in Figure J.18. Carries begin rippling



**Figure J.18 Carry-skip adder.** This is a 20-bit carry-skip adder ( $n=20$ ) with each block 4 bits wide ( $k=4$ ).



**Figure J.19 Combination of CLA and ripple-carry adder.** In the top row, carries ripple within each group of four boxes.

simultaneously through each block. If any block generates a carry, then the carry-out of a block will be true, even though the carry-in to the block may not be correct yet. If at the start of each add operation the carry-in to each block is 0, then no spurious carry-outs will be generated. Thus, the carry-out of each block can be thought of as if it were the  $G$  signal. Once the carry-out from the least-significant block is generated, it not only feeds into the next block but is also fed through the AND gate with the  $P$  signal from that next block. If the carry-out and  $P$  signals are both true, then the carry *skips* the second block and is ready to feed into the third block, and so on. The carry-skip adder is only practical if the carry-in signals can be easily cleared at the start of each operation—for example, by precharging in CMOS.

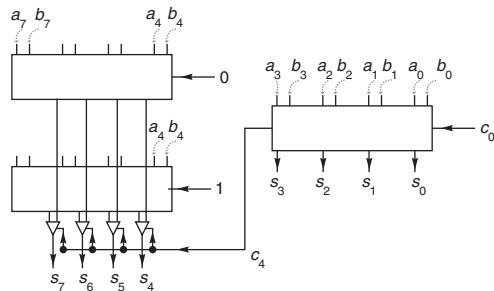
To analyze the speed of a carry-skip adder, let's assume that it takes 1 time unit for a signal to pass through two logic levels. Then it will take  $k$  time units for a carry to ripple across a block of size  $k$ , and it will take 1 time unit for a carry to skip a block. The longest signal path in the carry-skip adder starts with a carry being generated at the 0th position. If the adder is  $n$  bits wide, then it takes  $k$  time units to ripple through the first block,  $n/k - 2$  time units to skip blocks, and  $k$  more to ripple through the last block. To be specific: if we have a 20-bit adder broken into groups of 4 bits, it will take  $4 + (20/4 - 2) + 4 = 11$  time units to perform an

add. Some experimentation reveals that there are more efficient ways to divide 20 bits into blocks. For example, consider five blocks with the least-significant 2 bits in the first block, the next 5 bits in the second block, followed by blocks of size 6, 5, and 2. Then the add time is reduced to 9 time units. This illustrates an important general principle. For a carry-skip adder, making the interior blocks larger will speed up the adder. In fact, the same idea of varying the block sizes can sometimes speed up other adder designs as well. Because of the large amount of rippling, a carry-skip adder is most appropriate for technologies where rippling is fast.

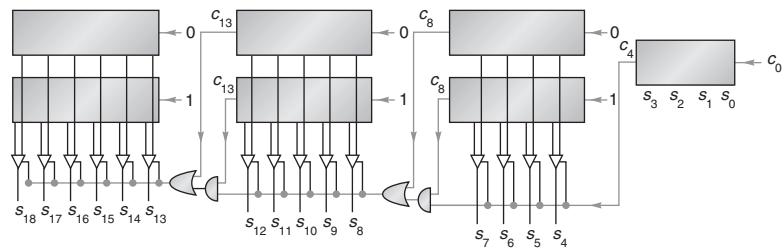
### Carry-Select Adder

A *carry-select adder* works on the following principle: Two additions are performed in parallel, one assuming the carry-in is 0 and the other assuming the carry-in is 1. When the carry-in is finally known, the correct sum (which has been precomputed) is simply selected. An example of such a design is shown in Figure J.20. An 8-bit adder is divided into two halves, and the carry-out from the lower half is used to select the sum bits from the upper half. If each block is computing its sum using rippling (a linear time algorithm), then the design in Figure J.20 is twice as fast at 50% more cost. However, note that the  $c_4$  signal must drive many muxes, which may be very slow in some technologies. Instead of dividing the adder into halves, it could be divided into quarters for a still further speedup. This is illustrated in Figure J.21. If it takes  $k$  time units for a block to add  $k$ -bit numbers, and if it takes 1 time unit to compute the mux input from the two carry-out signals, then for optimal operation each block should be 1 bit wider than the next, as shown in Figure J.21. Therefore, as in the carry-skip adder, the best design involves variable-size blocks.

As a summary of this section, the asymptotic time and space requirements for the different adders are given in Figure J.22. (The times for carry-skip and



**Figure J.20 Simple carry-select adder.** At the same time that the sum of the low-order 4 bits is being computed, the high-order bits are being computed twice in parallel: once assuming that  $c_4=0$  and once assuming  $c_4=1$ .



**Figure J.21** Carry-select adder. As soon as the carry-out of the rightmost block is known, it is used to select the other sum bits.

| Adder        | Time          | Space         |
|--------------|---------------|---------------|
| Ripple       | $O(n)$        | $O(n)$        |
| CLA          | $O(\log n)$   | $O(n \log n)$ |
| Carry-skip   | $O(\sqrt{n})$ | $O(n)$        |
| Carry-select | $O(\sqrt{n})$ | $O(n)$        |

**Figure J.22** Asymptotic time and space requirements for four different types of adders.

carry-select come from a careful choice of block size. See Exercise J.26 for the carry-skip adder.) These different adders shouldn't be thought of as disjoint choices, but rather as building blocks to be used in constructing an adder. The utility of these different building blocks is highly dependent on the technology used. For example, the carry-select adder works well when a signal can drive many muxes, and the carry-skip adder is attractive in technologies where signals can be cleared at the start of each operation. Knowing the asymptotic behavior of adders is useful in understanding them, but relying too much on that behavior is a pitfall. The reason is that asymptotic behavior is only important as  $n$  grows very large. But  $n$  for an adder is the bits of precision, and double precision today is the same as it was 20 years ago—about 53 bits. Although it is true that as computers get faster, computations get longer—and thus have more rounding error, which in turn requires more precision—this effect grows very slowly with time.

## J.9

### Speeding Up Integer Multiplication and Division

The multiplication and division algorithms presented in Section J.2 are fairly slow, producing 1 bit per cycle (although that cycle might be a fraction of the CPU instruction cycle time). In this section, we discuss various techniques for higher-performance multiplication and division, including the division algorithm used in the Pentium chip.

## Shifting over Zeros

Although the technique of shifting over zeros is not currently used much, it is instructive to consider. It is distinguished by the fact that its execution time is operand dependent. Its lack of use is primarily attributable to its failure to offer enough speedup over bit-at-a-time algorithms. In addition, pipelining, synchronization with the CPU, and good compiler optimization are difficult with algorithms that run in variable time. In multiplication, the idea behind shifting over zeros is to add logic that detects when the low-order bit of the A register is 0 (see Figure J.2(a) on page J-4) and, if so, skips the addition step and proceeds directly to the shift step—hence the term *shifting over zeros*.

What about shifting for division? In nonrestoring division, an ALU operation (either an addition or subtraction) is performed at every step. There appears to be no opportunity for skipping an operation. But think about division this way: To compute  $a/b$ , subtract multiples of  $b$  from  $a$ , and then report how many subtractions were done. At each stage of the subtraction process the remainder must fit into the P register of Figure J.2(b) (page J-4). In the case when the remainder is a small positive number, you normally subtract  $b$ ; but suppose instead you only shifted the remainder and subtracted  $b$  the next time. As long as the remainder was sufficiently small (its high-order bit 0), after shifting it still would fit into the P register, and no information would be lost. However, this method does require changing the way we keep track of the number of times  $b$  has been subtracted from  $a$ . This idea usually goes under the name of *SRT division*, for Sweeney, Robertson, and Tocher, who independently proposed algorithms of this nature. The main extra complication of SRT division is that the quotient bits cannot be determined immediately from the sign of P at each step, as they can be in ordinary nonrestoring division.

More precisely, to divide  $a$  by  $b$  where  $a$  and  $b$  are  $n$ -bit numbers, load  $a$  and  $b$  into the A and B registers, respectively, of Figure J.2 (page J-4).

### SRT Division

1. If B has  $k$  leading zeros when expressed using  $n$  bits, shift all the registers left  $k$  bits.
2. For  $i=0, n-1$ ,
  - a) If the top three bits of P are equal, set  $q_i=0$  and shift (P,A) one bit left.
  - b) If the top three bits of P are not all equal and P is negative, set  $q_i=-1$  (also written as  $\bar{1}$ ), shift (P,A) one bit left, and add B.
  - c) Otherwise set  $q_i=1$ , shift (P,A) one bit left, and subtract B.
- End loop
3. If the final remainder is negative, correct the remainder by adding B, and correct the quotient by subtracting 1 from  $q_0$ . Finally, the remainder must be shifted  $k$  bits right, where  $k$  is the initial shift.

| P       | A           |                                                                                                                                   |
|---------|-------------|-----------------------------------------------------------------------------------------------------------------------------------|
| 00000   | 1000        | Divide $8 = 1000$ by $3 = 0011$ . B contains 0011.                                                                                |
| 00010   | 0000        | Step 1: B had two leading 0 s, so shift left by 2. B now contains 1100.                                                           |
|         |             | Step 2.1: Top three bits are equal. This is case (a), so                                                                          |
| 00100   | <b>0000</b> | set $q_0 = 0$ and shift.                                                                                                          |
|         |             | Step 2.2: Top three bits not equal and $P \geq 0$ is case (c), so                                                                 |
| 01000   | <b>0001</b> | set $q_1 = 1$ and shift.                                                                                                          |
| + 10100 |             | Subtract B.                                                                                                                       |
| 11100   | <b>0001</b> | Step 2.3: Top bits equal is case (a), so                                                                                          |
| 11000   | <b>0010</b> | set $q_2 = 0$ and shift.                                                                                                          |
|         |             | Step 2.4: Top three bits unequal is case (b), so                                                                                  |
| 10000   | <b>0101</b> | set $q_3 = -1$ and shift.                                                                                                         |
| + 01100 |             | Add B.                                                                                                                            |
| 11100   |             | Step 3. remainder is negative so restore it and subtract 1 from $q$ .                                                             |
| + 01100 |             |                                                                                                                                   |
| 01000   |             | Must undo the shift in step 1, so right-shift by 2 to get true remainder.<br>Remainder = 10, quotient = $010\bar{1} - 1 = 0010$ . |

**Figure J.23** SRT division of  $1000_2/0011_2$ . The quotient bits are shown in bold, using the notation 1 for  $-1$ .

A numerical example is given in Figure J.23. Although we are discussing integer division, it helps in explaining the algorithm to imagine the binary point just left of the most-significant bit. This changes Figure J.23 from  $01000_2/0011_2$  to  $0.1000_2/.0011_2$ . Since the binary point is changed in both the numerator and denominator, the quotient is not affected. The (P,A) register pair holds the remainder and is a two's complement number. For example, if P contains  $11110_2$  and A = 0, then the remainder is  $1.1110_2 = -1/8$ . If  $r$  is the value of the remainder, then  $-1 \leq r < 1$ .

Given these preliminaries, we can now analyze the SRT division algorithm. The first step of the algorithm shifts  $b$  so that  $b \geq 1/2$ . The rule for which ALU operation to perform is this: If  $-1/4 \leq r < 1/4$  (true whenever the top three bits of P are equal), then compute  $2r$  by shifting (P,A) left one bit; if  $r < 0$  (and hence  $r < -1/4$ , since otherwise it would have been eliminated by the first condition), then compute  $2r + b$  by shifting and then adding; if  $r \geq 1/4$  and subtract  $b$  from  $2r$ . Using  $b \geq 1/2$ , it is easy to check that these rules keep  $-1/2 \leq r < 1/2$ . For nonrestoring division, we only have  $|r| \leq b$ , and we need P to be  $n+1$  bits wide. But, for SRT division, the bound on  $r$  is tighter, namely,  $-1/2 \leq r < 1/2$ . Thus, we can save a bit by eliminating the high-order bit of P (and  $b$  and the adder). In particular, the test for equality of the top three bits of P becomes a test on just two bits.

The algorithm might change slightly in an implementation of SRT division. After each ALU operation, the P register can be shifted as many places as necessary to make either  $r \geq 1/4$  or  $r < -1/4$ . By shifting  $k$  places,  $k$  quotient bits are set equal to zero all at once. For this reason SRT division is sometimes described as one that keeps the remainder normalized to  $|r| \geq 1/4$ .

Notice that the value of the quotient bit computed in a given step is based on which operation is performed in that step (which in turn depends on the result of the operation from the previous step). This is in contrast to nonrestoring division, where the quotient bit computed in the  $i$ th step depends on the result of the operation in the same step. This difference is reflected in the fact that when the final remainder is negative, the last quotient bit must be adjusted in SRT division, but not in nonrestoring division. However, the key fact about the quotient bits in SRT division is that they can include  $\bar{1}$ . Although Figure J.23 shows the quotient bits being stored in the low-order bits of A, an actual implementation can't do this because you can't fit the three values  $-1, 0, 1$  into one bit. Furthermore, the quotient must be converted to ordinary two's complement in a full adder. A common way to do this is to accumulate the positive quotient bits in one register and the negative quotient bits in another, and then subtract the two registers after all the bits are known. Because there is more than one way to write a number in terms of the digits  $-1, 0, 1$ , SRT division is said to use a *redundant* quotient representation.

The differences between SRT division and ordinary nonrestoring division can be summarized as follows:

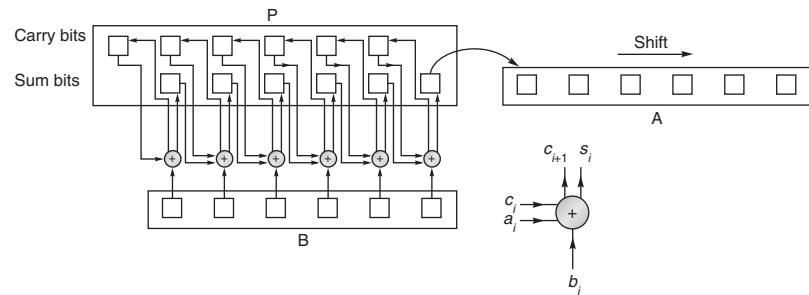
1. ALU decision rule—In nonrestoring division, it is determined by the sign of P; in SRT, it is determined by the two most-significant bits of P.
2. Final quotient—In nonrestoring division, it is immediate from the successive signs of P; in SRT, there are three quotient digits ( $1, 0, \bar{1}$ ), and the final quotient must be computed in a full  $n$ -bit adder.
3. Speed—SRT division will be faster on operands that produce zero quotient bits.

The simple version of the SRT division algorithm given above does not offer enough of a speedup to be practical in most cases. However, later on in this section we will study variants of SRT division that are quite practical.

## Speeding Up Multiplication with a Single Adder

As mentioned before, shifting-over-zero techniques are not used much in current hardware. We now discuss some methods that are in widespread use. Methods that increase the speed of multiplication can be divided into two classes: those that use a single adder and those that use multiple adders. Let's first discuss techniques that use a single adder.

In the discussion of addition we noted that, because of carry propagation, it is not practical to perform addition with two levels of logic. Using the cells of Figure J.17, adding two 64-bit numbers will require a trip through seven cells to compute the  $P$ 's and  $G$ 's and seven more to compute the carry bits, which will require at least 28 logic levels. In the simple multiplier of Figure J.2 on page J-4, each multiplication step passes through this adder. The amount of computation in each step can be dramatically reduced by using *carry-save adders* (CSAs). A carry-save adder is simply a collection of  $n$  independent full adders. A multiplier using



**Figure J.24 Carry-save multiplier.** Each circle represents a (3,2) adder working independently. At each step, the only bit of P that needs to be shifted is the low-order sum bit.

such an adder is illustrated in Figure J.24. Each circle marked “+” is a single-bit full adder, and each box represents one bit of a register. Each addition operation results in a pair of bits, stored in the sum and carry parts of P. Since each add is independent, only two logic levels are involved in the add—a vast improvement over 28.

To operate the multiplier in Figure J.24, load the sum and carry bits of P with zero and perform the first ALU operation. (If Booth recoding is used, it might be a subtraction rather than an addition.) Then shift the low-order sum bit of P into A, as well as shifting A itself. The  $n - 1$  high-order bits of P don’t need to be shifted because on the next cycle the sum bits are fed into the next lower-order adder. Each addition step is substantially increased in speed, since each add cell is working independently of the others, and no carry is propagated.

There are two drawbacks to carry-save adders. First, they require more hardware because there must be a copy of register P to hold the carry outputs of the adder. Second, after the last step, the high-order word of the result must be fed into an ordinary adder to combine the sum and carry parts. One way to accomplish this is by feeding the output of P into the adder used to perform the addition operation. Multiplying with a carry-save adder is sometimes called *redundant multiplication* because P is represented using two registers. Since there are many ways to represent P as the sum of two registers, this representation is redundant. The term *carry-propagate adder* (CPA) is used to denote an adder that is not a CSA. A propagate adder may propagate its carries using ripples, carry-lookahead, or some other method.

Another way to speed up multiplication without using extra adders is to examine  $k$  low-order bits of A at each step, rather than just one bit. This is often called *higher-radix multiplication*. As an example, suppose that  $k = 2$ . If the pair of bits is 00, add 0 to P; if it is 01, add B. If it is 10, simply shift b one bit left before adding it to P. Unfortunately, if the pair is 11, it appears we would have to compute  $b + 2b$ . But this can be avoided by using a higher-radix version of Booth recoding. Imagine A as a base 4 number: When the digit 3 appears, change it to  $\bar{1}$  and add 1 to the next higher digit to compensate. An extra benefit of using this scheme is that just like ordinary Booth recoding, it works for negative as well as positive integers (Section J.2).

The precise rules for radix-4 Booth recoding are given in Figure J.25. At the  $i$ th multiply step, the two low-order bits of the A register contain  $a_{2i}$  and  $a_{2i+1}$ . These two bits, together with the bit just shifted out ( $a_{2i-1}$ ), are used to select the multiple of  $b$  that must be added to the P register. A numerical example is given in Figure J.26. Another name for this multiplication technique is *overlapping triplets*, since it looks at 3 bits to determine what multiple of  $b$  to use, whereas ordinary Booth recoding looks at 2 bits.

Besides having more complex control logic, overlapping triplets also requires that the P register be 1 bit wider to accommodate the possibility of  $2b$  or  $-2b$  being added to it. It is possible to use a radix-8 (or even higher) version of Booth recoding. In that case, however, it would be necessary to use the multiple 3B as a potential summand. Radix-8 multipliers normally compute 3B once and for all at the beginning of a multiplication operation.

| Low-order bits of A |      | Last bit shifted out |          |
|---------------------|------|----------------------|----------|
| $2i+1$              | $2i$ | $2i-1$               | Multiple |
| 0                   | 0    | 0                    | 0        |
| 0                   | 0    | 1                    | $+b$     |
| 0                   | 1    | 0                    | $+b$     |
| 0                   | 1    | 1                    | $+2b$    |
| 1                   | 0    | 0                    | $-2b$    |
| 1                   | 0    | 1                    | $-b$     |
| 1                   | 1    | 0                    | $-b$     |
| 1                   | 1    | 1                    | 0        |

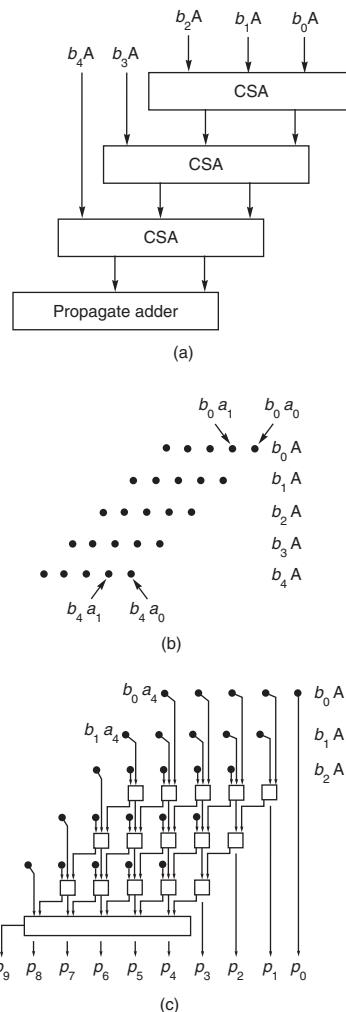
**Figure J.25 Multiples of  $b$  to use for radix-4 Booth recoding.** For example, if the two low-order bits of the A register are both 1, and the last bit to be shifted out of the A register is 0, then the correct multiple is  $-b$ , obtained from the second-to-last row of the table.

| P       | A    | L |                                                           |
|---------|------|---|-----------------------------------------------------------|
| 00000   | 1001 |   | Multiply $-7 = 1001$ times $-5 = 1011$ . B contains 1011. |
| + 11011 |      |   | Low-order bits of A are 0, 1; L=0, so add B.              |
| 11011   | 1001 |   |                                                           |
| 11110   | 1110 | 0 | Shift right by two bits, shifting in 1 s on the left.     |
| + 01010 |      |   | Low-order bits of A are 1, 0; L=0, so add $-2b$ .         |
| 01000   | 1110 | 0 |                                                           |
| 00010   | 0011 | 1 | Shift right by two bits.                                  |
|         |      |   | Product is $35 = 0100011$ .                               |

**Figure J.26 Multiplication of  $-7$  times  $-5$  using radix-4 Booth recoding.** The column labeled L contains the last bit shifted out the right end of A.

## Faster Multiplication with Many Adders

If the space for many adders is available, then multiplication speed can be improved. Figure J.27 shows a simple array multiplier for multiplying two 5-bit numbers, using three CSAs and one propagate adder. Part (a) is a block diagram of the kind we will use throughout this section. Parts (b) and (c) show the adder in more detail. All the inputs to the adder are shown in (b); the actual adders with their interconnections are shown in (c). Each row of adders in (c) corresponds to a box in



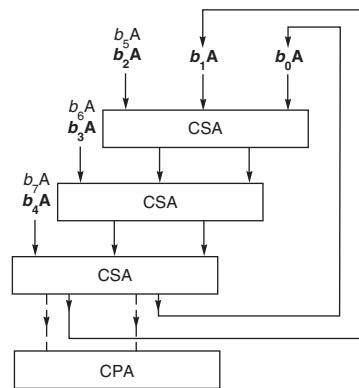
**Figure J.27** An array multiplier. The 5-bit number in A is multiplied by  $b_4b_3b_2b_1b_0$ . Part (a) shows the block diagram, (b) shows the inputs to the array, and (c) expands the array to show all the adders.

(a). The picture is “twisted” so that bits of the same significance are in the same column. In an actual implementation, the array would most likely be laid out as a square instead.

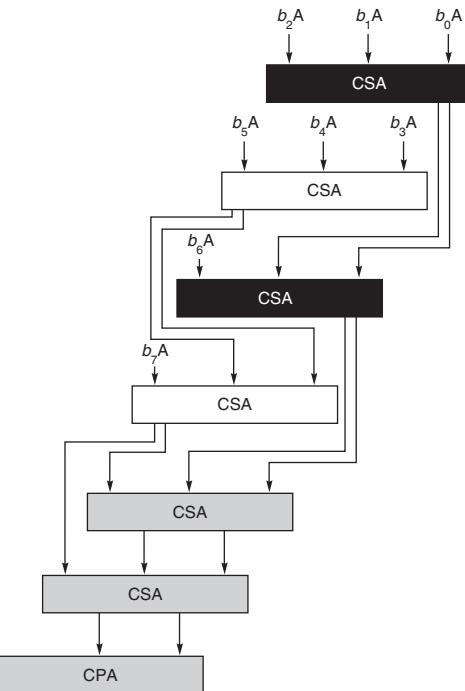
The array multiplier in Figure J.27 performs the same number of additions as the design in Figure J.24, so its latency is not dramatically different from that of a single carry-save adder. However, with the hardware in Figure J.27, multiplication can be pipelined, increasing the total throughput. On the other hand, although this level of pipelining is sometimes used in array processors, it is not used in any of the single-chip, floating-point accelerators discussed in Section J.10. Pipelining is discussed in general in Appendix C and by Kogge [1981] in the context of multipliers.

Sometimes the space budgeted on a chip for arithmetic may not hold an array large enough to multiply two double-precision numbers. In this case, a popular design is to use a two-pass arrangement such as the one shown in Figure J.28. The first pass through the array “retires” 5 bits of B. Then the result of this first pass is fed back into the top to be combined with the next three summands. The result of this second pass is then fed into a CPA. This design, however, loses the ability to be pipelined.

If arrays require as many addition steps as the much cheaper arrangements in Figures J.2 and J.24, why are they so popular? First of all, using an array has a smaller latency than using a single adder—because the array is a combinational circuit, the signals flow through it directly without being clocked. Although the two-pass adder of Figure J.28 would normally still use a clock, the cycle time for passing through  $k$  arrays can be less than  $k$  times the clock that would be needed for designs like the ones in Figures J.2 or J.24. Second, the array is amenable to various schemes for further speedup. One of them is shown in Figure J.29. The idea of this design is that two adds proceed in parallel or, to put it another way, each stream passes through only half the adders. Thus, it runs at almost twice



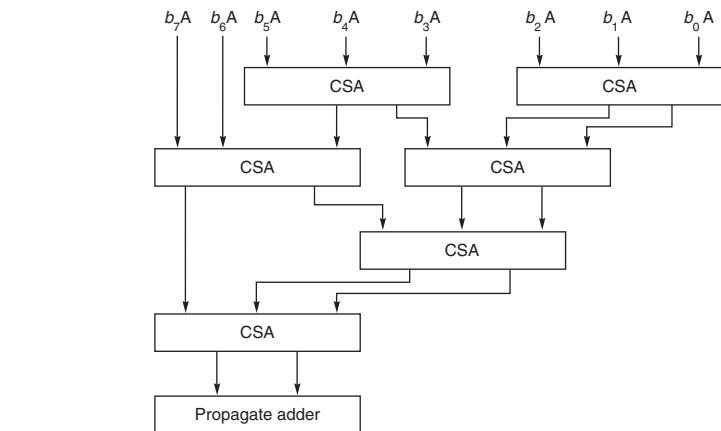
**Figure J.28 Multipass array multiplier.** Multiplies two 8-bit numbers with about half the hardware that would be used in a one-pass design like that of Figure J.27. At the end of the second pass, the bits flow into the CPA. The inputs used in the first pass are marked in bold.



**Figure J.29 Even/odd array.** The first two adders work in parallel. Their results are fed into the third and fourth adders, which also work in parallel, and so on.

the speed of the multiplier in Figure J.27. This *even/odd* multiplier is popular in VLSI because of its regular structure. Arrays can also be speeded up using asynchronous logic. One of the reasons why the multiplier of Figure J.2 (page J-4) needs a clock is to keep the output of the adder from feeding back into the input of the adder before the output has fully stabilized. Thus, if the array in Figure J.28 is long enough so that no signal can propagate from the top through the bottom in the time it takes for the first adder to stabilize, it may be possible to avoid clocks altogether. Williams et al. [1987] discussed a design using this idea, although it is for dividers instead of multipliers.

The techniques of the previous paragraph still have a multiply time of  $O(n)$ , but the time can be reduced to  $\log n$  using a tree. The simplest tree would combine pairs of summands  $b_0A \dots b_{n-1}A$ , cutting the number of summands from  $n$  to  $n/2$ . Then these  $n/2$  numbers would be added in pairs again, reducing to  $n/4$ , and so on, and resulting in a single sum after  $\log n$  steps. However, this simple binary tree idea doesn't map into full (3,2) adders, which reduce three inputs to two rather than reducing two inputs to one. A tree that does use full adders, known as a *Wallace tree*, is shown in Figure J.30. When computer arithmetic units were built out of



**Figure J.30 Wallace tree multiplier.** An example of a multiply tree that computes a product in  $O(\log n)$  steps.

MSI parts, a Wallace tree was the design of choice for high-speed multipliers. There is, however, a problem with implementing it in VLSI. If you try to fill in all the adders and paths for the Wallace tree of Figure J.30, you will discover that it does not have the nice, regular structure of Figure J.27. This is why VLSI designers have often chosen to use other  $\log n$  designs such as the *binary tree multiplier*, which is discussed next.

The problem with adding summands in a binary tree is coming up with a (2,1) adder that combines two digits and produces a single-sum digit. Because of carries, this isn't possible using binary notation, but it can be done with some other representation. We will use the *signed-digit representation* 1,  $\bar{1}$ , and 0, which we used previously to understand Booth's algorithm. This representation has two costs. First, it takes 2 bits to represent each signed digit. Second, the algorithm for adding two signed-digit numbers  $a_i$  and  $b_i$  is complex and requires examining  $a_i a_{i-1} a_{i-2}$  and  $b_i b_{i-1} b_{i-2}$ . Although this means you must look 2 bits back, in binary addition you might have to look an arbitrary number of bits back because of carries.

We can describe the algorithm for adding two signed-digit numbers as follows. First, compute sum and carry bits  $s_i$  and  $c_{i+1}$  using Figure J.31. Then compute the final sum as  $s_i + c_i$ . The tables are set up so that this final sum does not generate a carry.

---

**Example** What is the sum of the signed-digit numbers  $1\bar{1}0_2$  and  $001_2$ ?

**Answer** The two low-order bits sum to  $0 + 1 = 1\bar{1}$ , the next pair sums to  $\bar{1} + 0 = 0\bar{1}$ , and the high-order pair sums to  $1 + 0 = 01$ , so the sum is  $1\bar{1} + 0\bar{1}0 + 0100 = 10\bar{1}_2$ .

---

|                                                      |                                                                   |                                                                                         |                                                        |                                                                                                          |                                                                                               |
|------------------------------------------------------|-------------------------------------------------------------------|-----------------------------------------------------------------------------------------|--------------------------------------------------------|----------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------|
| $\begin{array}{r} 1 \\ + 1 \\ \hline 10 \end{array}$ | $\begin{array}{r} 1 \\ + \overline{1} \\ \hline 0\ 0 \end{array}$ | $\begin{array}{r} \overline{1} \\ + \overline{1} \\ \hline \overline{1}\ 0 \end{array}$ | $\begin{array}{r} 0 \\ + 0 \\ \hline 0\ 0 \end{array}$ | $\begin{array}{r} \overline{1}\ x \\ + \overline{0}\ y \\ \hline \overline{1}\ \overline{1} \end{array}$ | $\begin{array}{r} \overline{1}\ x \\ + 0\ y \\ \hline \overline{1}\ \overline{1} \end{array}$ |
|                                                      |                                                                   |                                                                                         |                                                        | $\begin{array}{l} \text{if } x \geq 0 \text{ and} \\ y \geq 0 \text{ otherwise} \end{array}$             | $\begin{array}{l} \text{if } x \geq 0 \text{ and} \\ y \geq 0 \text{ otherwise} \end{array}$  |

**Figure J.31 Signed-digit addition table.** The leftmost sum shows that when computing  $1+1$ , the sum bit is 0 and the carry bit is 1.

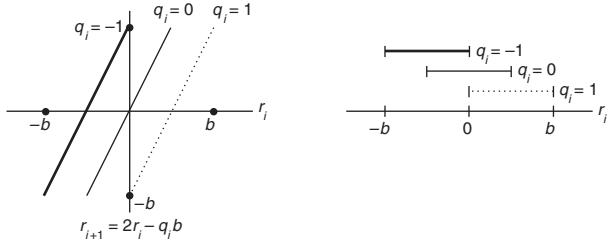
This, then, defines a (2,1) adder. With this in hand, we can use a straightforward binary tree to perform multiplication. In the first step it adds  $b_0A + b_1A$  in parallel with  $b_2A + b_3A, \dots, b_{n-2}A + b_{n-1}A$ . The next step adds the results of these sums in pairs, and so on. Although the final sum must be run through a carry-propagate adder to convert it from signed-digit form to two's complement, this final add step is necessary in any multiplier using CSAs.

To summarize, both Wallace trees and signed-digit trees are  $\log n$  multipliers. The Wallace tree uses fewer gates but is harder to lay out. The signed-digit tree has a more regular structure, but requires 2 bits to represent each digit and has more complicated add logic. As with adders, it is possible to combine different multiply techniques. For example, Booth recoding and arrays can be combined. In Figure J.27 instead of having each input be  $b_iA$ , we could have it be  $b_ib_{i-1}A$ . To avoid having to compute the multiple  $3b$ , we can use Booth recoding.

### Faster Division with One Adder

The two techniques we discussed for speeding up multiplication with a single adder were carry-save adders and higher-radix multiplication. However, there is a difficulty when trying to utilize these approaches to speed up nonrestoring division. If the adder in Figure J.2(b) on page J-4 is replaced with a carry-save adder, then P will be replaced with two registers, one for the sum bits and one for the carry bits (compare with the multiplier in Figure J.24). At the end of each cycle, the sign of P is uncertain (since P is the unevaluated sum of the two registers), yet it is the sign of P that is used to compute the quotient digit and decide the next ALU operation. When a higher radix is used, the problem is deciding what value to subtract from P. In the paper-and-pencil method, you have to guess the quotient digit. In binary division, there are only two possibilities. We were able to finesse the problem by initially guessing one and then adjusting the guess based on the sign of P. This doesn't work in higher radices because there are more than two possible quotient digits, rendering quotient selection potentially quite complicated: You would have to compute all the multiples of  $b$  and compare them to P.

Both the carry-save technique and higher-radix division can be made to work if we use a redundant quotient representation. Recall from our discussion of SRT division (page J-45) that by allowing the quotient digits to be  $-1, 0$ , or  $1$ , there is often a choice of which one to pick. The idea in the previous algorithm was to choose  $0$  whenever possible, because that meant an ALU operation could be



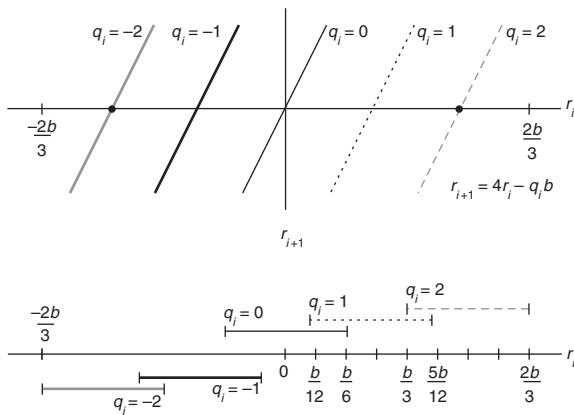
**Figure J.32 Quotient selection for radix-2 division.** The  $x$ -axis represents the  $i$ th remainder, which is the quantity in the (P,A) register pair. The  $y$ -axis shows the value of the remainder after one additional divide step. Each bar on the right-hand graph gives the range of  $r_i$  values for which it is permissible to select the associated value of  $q_i$ .

skipped. In carry-save division, the idea is that, because the remainder (which is the value of the (P,A) register pair) is not known exactly (being stored in carry-save form), the exact quotient digit is also not known. But, thanks to the redundant representation, the remainder doesn't have to be known precisely in order to pick a quotient digit. This is illustrated in Figure J.32, where the  $x$ -axis represents  $r_i$ , the remainder after  $i$  steps. The line labeled  $q_i=1$  shows the value that  $r_{i+1}$  would be if we chose  $q_i=1$ , and similarly for the lines  $q_i=0$  and  $q_i=-1$ . We can choose any value for  $q_i$ , as long as  $r_{i+1}=2r_i-q_ib$  satisfies  $|r_{i+1}| \leq b$ . The allowable ranges are shown in the right half of Figure J.32. This shows that you don't need to know the precise value of  $r_i$  in order to choose a quotient digit  $q_i$ . You only need to know that  $r$  lies in an interval small enough to fit entirely within one of the overlapping bars shown in the right half of Figure J.32.

This is the basis for using carry-save adders. Look at the high-order bits of the carry-save adder and sum them in a propagate adder. Then use this approximation of  $r$  (together with the divisor,  $b$ ) to compute  $q_i$ , usually by means of a lookup table. The same technique works for higher-radix division (whether or not a carry-save adder is used). The high-order bits P can be used to index a table that gives one of the allowable quotient digits.

The design challenge when building a high-speed SRT divider is figuring out how many bits of P and B need to be examined. For example, suppose that we take a radix of 4, use quotient digits of 2, 1, 0,  $\bar{1}$ ,  $\bar{2}$ , but have a propagate adder. How many bits of P and B need to be examined? Deciding this involves two steps. For ordinary radix-2 nonrestoring division, because at each stage  $|r| \leq b$ , the P buffer won't overflow. But, for radix 4,  $r_{i+1} = 4r_i - q_ib$  is computed at each stage, and if  $r_i$  is near  $b$ , then  $4r_i$  will be near  $4b$ , and even the largest quotient digit will not bring  $r$  back to the range  $|r_{i+1}| \leq b$ . In other words, the remainder might grow without bound. However, restricting  $|r_i| \leq 2b/3$  makes it easy to check that  $r_i$  will stay bounded.

After figuring out the bound that  $r_i$  must satisfy, we can draw the diagram in Figure J.33, which is analogous to Figure J.32. For example, the diagram shows

Figure J.33 Quotient selection for radix-4 division with quotient digits  $-2, -1, 0, 1, 2$ .

that if  $r_i$  is between  $(1/12)b$  and  $(5/12)b$ , we can pick  $q=1$ , and so on. Or, to put it another way, if  $r/b$  is between  $1/12$  and  $5/12$ , we can pick  $q=1$ . Suppose the divider examines 5 bits of P (including the sign bit) and 4 bits of b (ignoring the sign, since it is always nonnegative). The interesting case is when the high bits of P are 00011xxx..., while the high bits of b are 1001xxx.... Imagine the binary point at the left end of each register. Since we truncated, r (the value of P concatenated with A) could have a value from  $0.0011_2$  to  $0.0100_2$ , and b could have a value from  $.1001_2$  to  $.1010_2$ . Thus,  $r/b$  could be as small as  $0.0011_2/.1010_2$  or as large as  $0.0100_2/.1001_2$ , but  $0.0011_2/.1010_2 = 3/10 < 1/3$  would require a quotient bit of 1, while  $0.0100_2/.1001_2 = 4/9 > 5/12$  would require a quotient bit of 2. In other words, 5 bits of P and 4 bits of b aren't enough to pick a quotient bit. It turns out that 6 bits of P and 4 bits of b are enough. This can be verified by writing a simple program that checks all the cases. The output of such a program is shown in Figure J.34.

---

**Example** Using 8-bit registers, compute  $149/5$  using radix-4 SRT division.

**Answer** Follow the SRT algorithm on page J-45, but replace the quotient selection rule in step 2 with one that uses Figure J.34. See Figure J.35.

---

The Pentium uses a radix-4 SRT division algorithm like the one just presented, except that it uses a carry-save adder. Exercises J.34(c) and J.35 explore this in detail. Although these are simple cases, all SRT analyses proceed in the same way. First compute the range of  $r_i$ , then plot  $r_i$  against  $r_{i+1}$  to find the quotient ranges, and finally write a program to compute how many bits are necessary. (It is sometimes also possible to compute the required number of bits analytically.) Various details need to be considered in building a practical SRT divider.

| $b$ | Range of P |    |    | $q$ | $b$ | Range of P |     |    | $q$ |
|-----|------------|----|----|-----|-----|------------|-----|----|-----|
| 8   | -12        | -7 | -2 | -2  | 12  | -18        | -10 | -2 |     |
| 8   | -6         | -3 | -1 | -1  | 12  | -10        | -4  | -1 |     |
| 8   | -2         | 1  | 0  | 0   | 12  | -4         | 3   | 0  |     |
| 8   | 2          | 5  | 1  | 1   | 12  | 3          | 9   | 1  |     |
| 8   | 6          | 11 | 2  | 2   | 12  | 9          | 17  | 2  |     |
| 9   | -14        | -8 | -2 | -2  | 13  | -19        | -11 | -2 |     |
| 9   | -7         | -3 | -1 | -1  | 13  | -10        | -4  | -1 |     |
| 9   | -3         | 2  | 0  | 0   | 13  | -4         | 3   | 0  |     |
| 9   | 2          | 6  | 1  | 1   | 13  | 3          | 9   | 1  |     |
| 9   | 7          | 13 | 2  | 2   | 13  | 10         | 18  | 2  |     |
| 10  | -15        | -9 | -2 | -2  | 14  | -20        | -11 | -2 |     |
| 10  | -8         | -3 | -1 | -1  | 14  | -11        | -4  | -1 |     |
| 10  | -3         | 2  | 0  | 0   | 14  | -4         | 3   | 0  |     |
| 10  | 2          | 7  | 1  | 1   | 14  | 3          | 10  | 1  |     |
| 10  | 8          | 14 | 2  | 2   | 14  | 10         | 19  | 2  |     |
| 11  | -16        | -9 | -2 | -2  | 15  | -22        | -12 | -2 |     |
| 11  | -9         | -3 | -1 | -1  | 15  | -12        | -4  | -1 |     |
| 11  | -3         | 2  | 0  | 0   | 15  | -5         | 4   | 0  |     |
| 11  | 2          | 8  | 1  | 1   | 15  | 3          | 11  | 1  |     |
| 11  | 8          | 15 | 2  | 2   | 15  | 11         | 21  | 2  |     |

**Figure J.34 Quotient digits for radix-4 SRT division with a propagate adder.** The top row says that if the high-order 4 bits of  $b$  are  $1000_2=8$ , and if the top 6 bits of P are between  $110100_2=-12$  and  $111001_2=-7$ , then -2 is a valid quotient digit.

For example, the quotient lookup table has a fairly regular structure, which means it is usually cheaper to encode it as a PLA rather than in ROM. For more details about SRT division, see Burgess and Williams [1995].

## J.10

## Putting It All Together

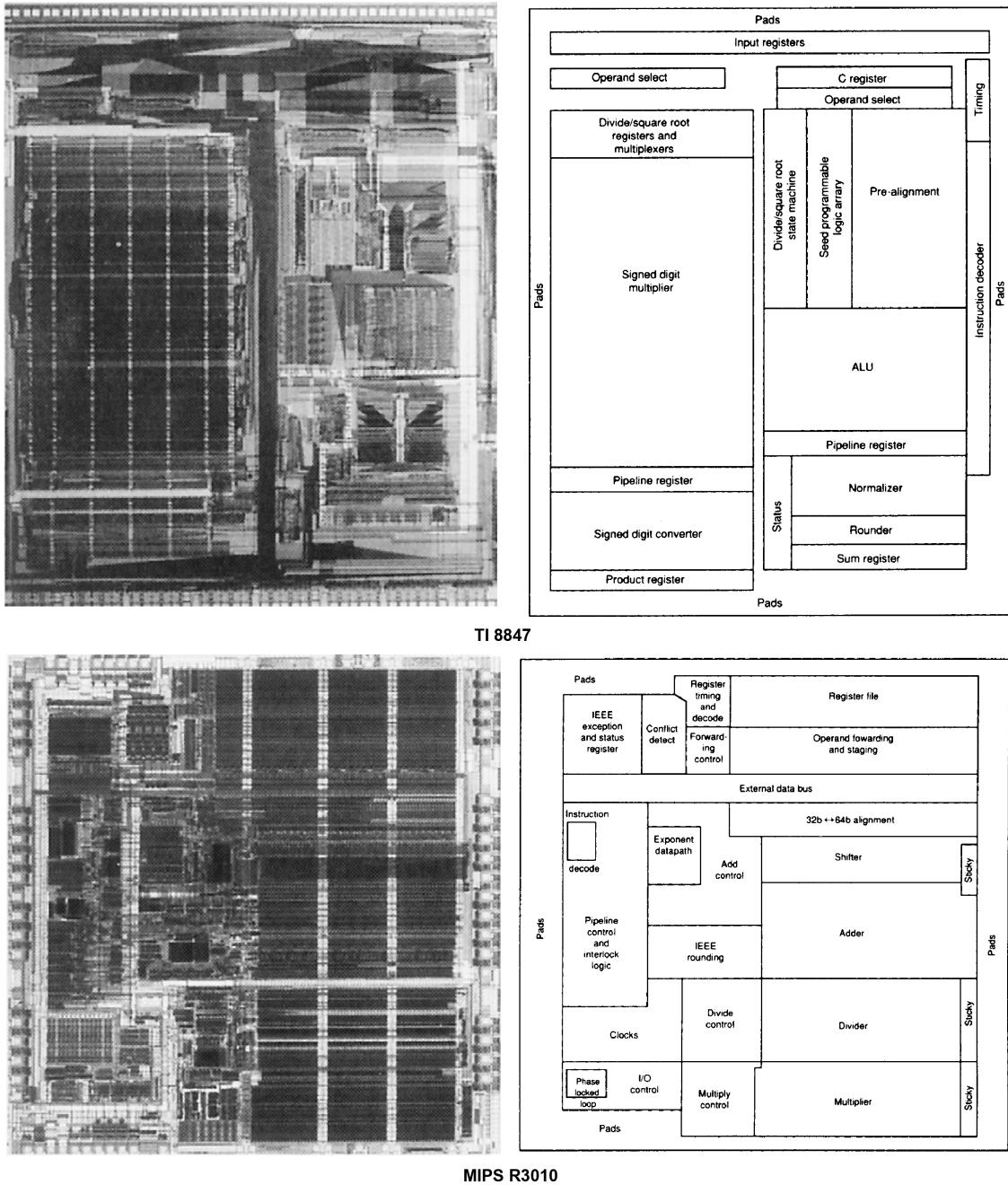
In this section, we will compare the Weitek 3364, the MIPS R3010, and the Texas Instruments 8847 (see Figures J.36 and J.37). In many ways, these are ideal chips to compare. They each implement the IEEE standard for addition, subtraction,

| P           | A        |                                                                                                              |
|-------------|----------|--------------------------------------------------------------------------------------------------------------|
| 000000000   | 10010101 | Divide 149 by 5. B contains 00000101.                                                                        |
| 000010010   | 10100000 | Step 1: B had 5 leading 0s, so shift left by 5. B now contains 10100000, so use $b=10$ section of table.     |
| 001001010   | 1000000  | Step 2.1: Top 6 bits of P are 2, so shift left by 2. From table, can pick $q$ to be 0 or 1. Choose $q_0=0$ . |
| 100101010   | 000002   | Step 2.2: Top 6 bits of P are 9, so shift left 2. $q_1=2$ .                                                  |
| + 011000000 |          | Subtract $2b$ .                                                                                              |
| 111101010   | 000002   | Step 2.3: Top bits = -3, so shift left 2. Can pick 0 or -1 for $q$ , pick $q_2=0$ .                          |
| 110101000   | 00020    | Step 2.4: Top bits = -11, so shift left 2. $q_3=-2$ .                                                        |
| 010100000   | 0202     |                                                                                                              |
| + 101000000 |          | Add $2b$ .                                                                                                   |
| 111100000   |          | Step 3: Remainder is negative, so restore by adding $b$ and subtract 1 from $q$ .                            |
| + 010100000 |          |                                                                                                              |
| 010000000   |          | Answer: $q = 020\bar{2} - 1 = 29$                                                                            |
|             |          | To get remainder, undo shift in step 1 so remainder = 010000000 > 5 = 4.                                     |

**Figure J.35 Example of radix-4 SRT division.** Division of 149 by 5.

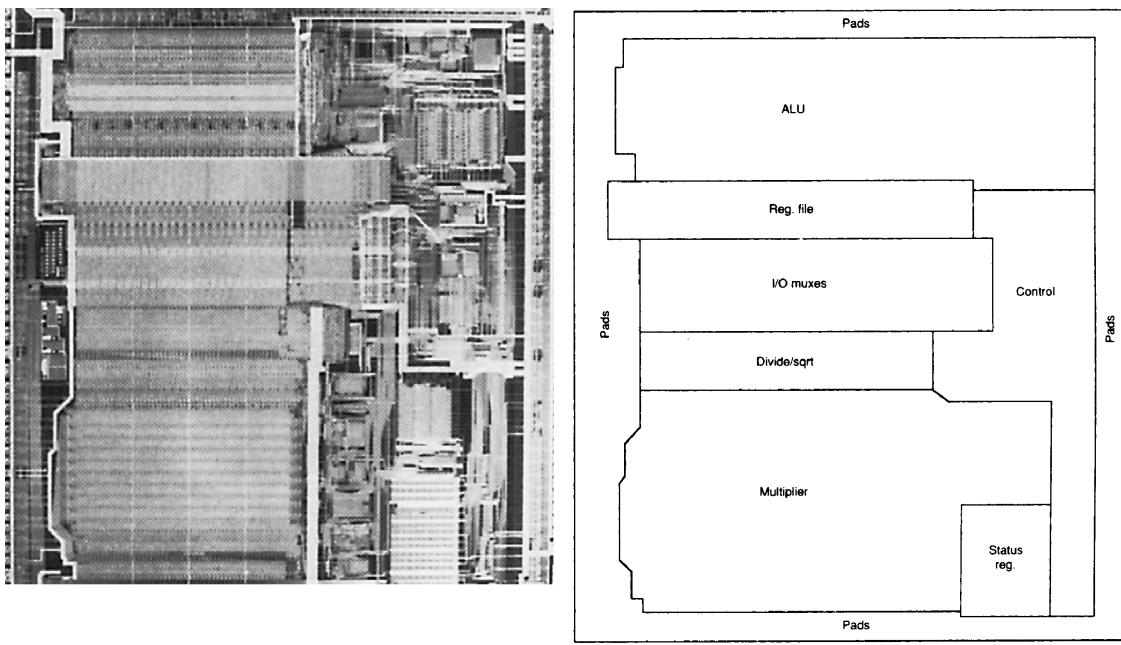
| Features                 | MIPS R3010 | Weitek 3364 | TI 8847 |
|--------------------------|------------|-------------|---------|
| Clock cycle time (ns)    | 40         | 50          | 30      |
| Size (mil <sup>2</sup> ) | 114,857    | 147,600     | 156,180 |
| Transistors              | 75,000     | 165,000     | 180,000 |
| Pins                     | 84         | 168         | 207     |
| Power (watts)            | 3.5        | 1.5         | 1.5     |
| Cycles/add               | 2          | 2           | 2       |
| Cycles/mult              | 5          | 2           | 3       |
| Cycles/divide            | 19         | 17          | 11      |
| Cycles/square root       | —          | 30          | 14      |

**Figure J.36 Summary of the three floating-point chips discussed in this section.** The cycle times are for production parts available in June 1989. The cycle counts are for double-precision operations.



**Figure J.37** Chip layout for the TI 8847, MIPS R3010, and Weitek 3364. In the left-hand columns are the photo-micrographs; the right-hand columns show the corresponding floor plans.

(Continued)



Weitek 3364

Figure J.37 (Continued)

multiplication, and division on a single chip. All were introduced in 1988 and run with a cycle time of about 40 nanoseconds. However, as we will see, they use quite different algorithms. The Weitek chip is well described in Birman et al. [1990], the MIPS chip is described in less detail in Rowen, Johnson, and Ries [1988], and details of the TI chip can be found in Darley et al. [1989].

These three chips have a number of things in common. They perform addition and multiplication in parallel, and they implement neither extended precision nor a remainder step operation. (Recall from Section J.6 that it is easy to implement the IEEE remainder function in software if a remainder step instruction is available.) The designers of these chips probably decided not to provide extended precision because the most influential users are those who run portable codes, which can't rely on extended precision. However, as we have seen, extended precision can make for faster and simpler math libraries.

In the summary of the three chips given in Figure J.36, note that a higher transistor count generally leads to smaller cycle counts. Comparing the cycles/op numbers needs to be done carefully, because the figures for the MIPS chip are those for a complete system (R3000/3010 pair), while the Weitek and TI numbers are for stand-alone chips and are usually larger when used in a complete system.

The MIPS chip has the fewest transistors of the three. This is reflected in the fact that it is the only chip of the three that does not have any pipelining or hardware

square root. Further, the multiplication and addition operations are not completely independent because they share the carry-propagate adder that performs the final rounding (as well as the rounding logic).

Addition on the R3010 uses a mixture of ripple, CLA, and carry-select. A carry-select adder is used in the fashion of Figure J.20 (page J-43). Within each half, carries are propagated using a hybrid ripple-CLA scheme of the type indicated in Figure J.19 (page J-42). However, this is further tuned by varying the size of each block, rather than having each fixed at 4 bits (as they are in Figure J.19). The multiplier is midway between the designs of Figures J.2 (page J-4) and J.27 (page J-50). It has an array just large enough so that output can be fed back into the input without having to be clocked. Also, it uses radix-4 Booth recoding and the even/odd technique of Figure J.29 (page J-52). The R3010 can do a divide and multiply in parallel (like the Weitek chip but unlike the TI chip). The divider is a radix-4 SRT method with quotient digits  $-2, -1, 0, 1$ , and  $2$ , and is similar to that described in Taylor [1985]. Double-precision division is about four times slower than multiplication. The R3010 shows that for chips using an  $O(n)$  multiplier, an SRT divider can operate fast enough to keep a reasonable ratio between multiply and divide.

The Weitek 3364 has independent add, multiply, and divide units. It also uses radix-4 SRT division. However, the add and multiply operations on the Weitek chip are pipelined. The three addition stages are (1) exponent compare, (2) add followed by shift (or *vice versa*), and (3) final rounding. Stages (1) and (3) take only a half-cycle, allowing the whole operation to be done in two cycles, even though there are three pipeline stages. The multiplier uses an array of the style of Figure J.28 but uses radix-8 Booth recoding, which means it must compute 3 times the multiplier. The three multiplier pipeline stages are (1) compute  $3b$ , (2) pass through array, and (3) final carry-propagation add and round. Single precision passes through the array once, double precision twice. Like addition, the latency is two cycles.

The Weitek chip uses an interesting addition algorithm. It is a variant on the carry-skip adder pictured in Figure J.18 (page J-42). However,  $P_{ij}$ , which is the logical AND of many terms, is computed by rippling, performing one AND per ripple. Thus, while the carries propagate left within a block, the value of  $P_{ij}$  is propagating right within the next block, and the block sizes are chosen so that both waves complete at the same time. Unlike the MIPS chip, the 3364 has hardware square root, which shares the divide hardware. The ratio of double-precision multiply to divide is 2:17. The large disparity between multiply and divide is due to the fact that multiplication uses radix-8 Booth recoding, while division uses a radix-4 method. In the MIPS R3010, multiplication and division use the same radix.

The notable feature of the TI 8847 is that it does division by iteration (using the Goldschmidt algorithm discussed in Section J.6). This improves the speed of division (the ratio of multiply to divide is 3:11), but means that multiplication and division cannot be done in parallel as on the other two chips. Addition has a two-stage pipeline. Exponent compare, fraction shift, and fraction addition are done in the first stage, normalization and rounding in the second stage. Multiplication uses

a binary tree of signed-digit adders and has a three-stage pipeline. The first stage passes through the array, retiring half the bits; the second stage passes through the array a second time; and the third stage converts from signed-digit form to two's complement. Since there is only one array, a new multiply operation can only be initiated in every other cycle. However, by slowing down the clock, two passes through the array can be made in a single cycle. In this case, a new multiplication can be initiated in each cycle. The 8847 adder uses a carry-select algorithm rather than carry-lookahead. As mentioned in Section J.6, the TI carries 60 bits of precision in order to do correctly rounded division.

These three chips illustrate the different trade-offs made by designers with similar constraints. One of the most interesting things about these chips is the diversity of their algorithms. Each uses a different add algorithm, as well as a different multiply algorithm. In fact, Booth recoding is the only technique that is universally used by all the chips.

## J.11

### Fallacies and Pitfalls

**Fallacy** *Underflows rarely occur in actual floating-point application code*

Although most codes rarely underflow, there are actual codes that underflow frequently. SDRWAVE [Kahaner 1988], which solves a one-dimensional wave equation, is one such example. This program underflows quite frequently, even when functioning properly. Measurements on one machine show that adding hardware support for gradual underflow would cause SDRWAVE to run about 50% faster.

**Fallacy** *Conversions between integer and floating point are rare*

In fact, in spice they are as frequent as divides. The assumption that conversions are rare leads to a mistake in the SPARC version 8 instruction set, which does not provide an instruction to move from integer registers to floating-point registers.

**Pitfall** *Don't increase the speed of a floating-point unit without increasing its memory bandwidth*

A typical use of a floating-point unit is to add two vectors to produce a third vector. If these vectors consist of double-precision numbers, then each floating-point add will use three operands of 64 bits each, or 24 bytes of memory. The memory bandwidth requirements are even greater if the floating-point unit can perform addition and multiplication in parallel (as most do).

**Pitfall**  *$-x$  is not the same as  $0 - x$*

This is a fine point in the IEEE standard that has tripped up some designers. Because floating-point numbers use the sign magnitude system, there are two zeros,  $+0$  and  $-0$ . The standard says that  $0 - 0 = +0$ , whereas  $- (0) = -0$ . Thus,  $-x$  is not the same as  $0 - x$  when  $x = 0$ .

**J.12****Historical Perspective and References**

The earliest computers used fixed point rather than floating point. In “Preliminary Discussion of the Logical Design of an Electronic Computing Instrument,” Burks, Goldstine, and von Neumann [1946] put it like this:

*There appear to be two major purposes in a “floating” decimal point system both of which arise from the fact that the number of digits in a word is a constant fixed by design considerations for each particular machine. The first of these purposes is to retain in a sum or product as many significant digits as possible and the second of these is to free the human operator from the burden of estimating and inserting into a problem “scale factors”—multiplicative constants which serve to keep numbers within the limits of the machine.*

*There is, of course, no denying the fact that human time is consumed in arranging for the introduction of suitable scale factors. We only argue that the time so consumed is a very small percentage of the total time we will spend in preparing an interesting problem for our machine. The first advantage of the floating point is, we feel, somewhat illusory. In order to have such a floating point, one must waste memory capacity that could otherwise be used for carrying more digits per word. It would therefore seem to us not at all clear whether the modest advantages of a floating binary point offset the loss of memory capacity and the increased complexity of the arithmetic and control circuits.*

This enables us to see things from the perspective of early computer designers, who believed that saving computer time and memory were more important than saving programmer time.

The original papers introducing the Wallace tree, Booth recoding, SRT division, overlapped triplets, and so on are reprinted in Swartzlander [1990]. A good explanation of an early machine (the IBM 360/91) that used a pipelined Wallace tree, Booth recoding, and iterative division is in Anderson et al. [1967]. A discussion of the average time for single-bit SRT division is in Freiman [1961]; this is one of the few interesting historical papers that does not appear in Swartzlander.

The standard book of Mead and Conway [1980] discouraged the use of CLAs as not being cost effective in VLSI. The important paper by Brent and Kung [1982] helped combat that view. An example of a detailed layout for CLAs can be found in Ngai and Irwin [1985] or in Weste and Eshraghian [1993], and a more theoretical treatment is given by Leighton [1992]. Takagi, Yasuura, and Yajima [1985] provide a detailed description of a signed-digit tree multiplier.

Before the ascendancy of IEEE arithmetic, many different floating-point formats were in use. Three important ones were used by the IBM 370, the DEC VAX, and the Cray. Here is a brief summary of these older formats. The VAX format is closest to the IEEE standard. Its single-precision format (F format) is like IEEE single precision in that it has a hidden bit, 8 bits of exponent, and 23 bits of fraction. However, it does not have a sticky bit, which causes it to round halfway cases up instead of to even. The VAX has a slightly different exponent range from IEEE

single:  $E_{\min}$  is  $-128$  rather than  $-126$  as in IEEE, and  $E_{\max}$  is  $126$  instead of  $127$ . The main differences between VAX and IEEE are the lack of special values and gradual underflow. The VAX has a reserved operand, but it works like a signaling NaN: It traps whenever it is referenced. Originally, the VAX's double precision (D format) also had 8 bits of exponent. However, as this is too small for many applications, a G format was added; like the IEEE standard, this format has 11 bits of exponent. The VAX also has an H format, which is 128 bits long.

The IBM 370 floating-point format uses base 16 rather than base 2. This means it cannot use a hidden bit. In single precision, it has 7 bits of exponent and 24 bits (6 hex digits) of fraction. Thus, the largest representable number is  $16^{27} = 2^4 \times 2^7 = 2^{29}$ , compared with  $2^{28}$  for IEEE. However, a number that is normalized in the hexadecimal sense only needs to have a nonzero leading digit. When interpreted in binary, the three most-significant bits could be zero. Thus, there are potentially fewer than 24 bits of significance. The reason for using the higher base was to minimize the amount of shifting required when adding floating-point numbers. However, this is less significant in current machines, where the floating-point add time is usually fixed independently of the operands. Another difference between 370 arithmetic and IEEE arithmetic is that the 370 has neither a round digit nor a sticky digit, which effectively means that it truncates rather than rounds. Thus, in many computations, the result will systematically be too small. Unlike the VAX and IEEE arithmetic, every bit pattern is a valid number. Thus, library routines must establish conventions for what to return in case of errors. In the IBM FORTRAN library, for example,  $\sqrt{-4}$  returns 2!

Arithmetic on Cray computers is interesting because it is driven by a motivation for the highest possible floating-point performance. It has a 15-bit exponent field and a 48-bit fraction field. Addition on Cray computers does not have a guard digit, and multiplication is even less accurate than addition. Thinking of multiplication as a sum of  $p$  numbers, each  $2p$  bits long, Cray computers drop the low-order bits of each summand. Thus, analyzing the exact error characteristics of the multiply operation is not easy. Reciprocals are computed using iteration, and division of  $a$  by  $b$  is done by multiplying  $a$  times  $1/b$ . The errors in multiplication and reciprocation combine to make the last three bits of a divide operation unreliable. At least Cray computers serve to keep numerical analysts on their toes!

The IEEE standardization process began in 1977, inspired mainly by W. Kahan and based partly on Kahan's work with the IBM 7094 at the University of Toronto [Kahan 1968]. The standardization process was a lengthy affair, with gradual underflow causing the most controversy. (According to Cleve Moler, visitors to the United States were advised that the sights not to be missed were Las Vegas, the Grand Canyon, and the IEEE standards committee meeting.) The standard was finally approved in 1985. The Intel 8087 was the first major commercial IEEE implementation and appeared in 1981, before the standard was finalized. It contains features that were eliminated in the final standard, such as projective bits. According to Kahan, the length of double-extended precision was based on what could be implemented in the 8087. Although the IEEE standard was not based on any existing floating-point system, most of its features were present in some other system. For example, the CDC 6600 reserved special bit patterns for INDEFINITE

and INFINITY, while the idea of denormal numbers appears in Goldberg [1967] as well as in Kahan [1968]. Kahan was awarded the 1989 Turing prize in recognition of his work on floating point.

Although floating point rarely attracts the interest of the general press, newspapers were filled with stories about floating-point division in November 1994. A bug in the division algorithm used on all of Intel's Pentium chips had just come to light. It was discovered by Thomas Nicely, a math professor at Lynchburg College in Virginia. Nicely found the bug when doing calculations involving reciprocals of prime numbers. News of Nicely's discovery first appeared in the press on the front page of the November 7 issue of *Electronic Engineering Times*. Intel's immediate response was to stonewall, asserting that the bug would only affect theoretical mathematicians. Intel told the press, "This doesn't even qualify as an errata ... even if you're an engineer, you're not going to see this."

Under more pressure, Intel issued a white paper, dated November 30, explaining why they didn't think the bug was significant. One of their arguments was based on the fact that if you pick two floating-point numbers at random and divide one into the other, the chance that the resulting quotient will be in error is about 1 in 9 billion. However, Intel neglected to explain why they thought that the typical customer accessed floating-point numbers randomly.

Pressure continued to mount on Intel. One sore point was that Intel had known about the bug before Nicely discovered it, but had decided not to make it public. Finally, on December 20, Intel announced that they would unconditionally replace any Pentium chip that used the faulty algorithm and that they would take an unspecified charge against earnings, which turned out to be \$300 million.

The Pentium uses a simple version of SRT division as discussed in Section J.9. The bug was introduced when they converted the quotient lookup table to a PLA. Evidently there were a few elements of the table containing the quotient digit 2 that Intel thought would never be accessed, and they optimized the PLA design using this assumption. The resulting PLA returned 0 rather than 2 in these situations. However, those entries were really accessed, and this caused the division bug. Even though the effect of the faulty PLA was to cause 5 out of 2048 table entries to be wrong, the Pentium only computes an incorrect quotient 1 out of 9 billion times on random inputs. This is explored in Exercise J.34.

## References

- Anderson, S.F., Earle, J.G., Goldschmidt, R.E., Powers, D.M., 1967. The IBM System/360 Model 91: Floating-point execution unit. *IBM J. Research and Development* 11, 34–53. Reprinted in Swartzlander [1990]. *Good description of an early high-performance floating-point unit that used a pipelined Wallace tree multiplier and iterative division.*
- Bell, C.G., Newell, A., 1971. Computer Structures: Readings and Examples. McGraw-Hill, New York.
- Birman, M., Samuels, A., Chu, G., Chuk, T., Hu, L., McLeod, J., Barnes, J., 1990. Developing the WRL3170/3171 SPARC floating-point coprocessors. *IEEE Micro* 10 (1), 55–64. *These chips have the same floating-point core as the Weitek 3364, and this paper has a fairly detailed description of that floating-point design.*

- Brent, R.P., Kung, H.T., 1982. A regular layout for parallel adders. IEEE Trans. on Computers C-31, 260–264. *This is the paper that popularized CLAs in VLSI.*
- Burgess, N., Williams, T., 1995. Choices of operand truncation in the SRT division algorithm. IEEE Trans. on Computers 44, 7. *Analyzes how many bits of divisor and remainder need to be examined in SRT division.*
- Burks, A.W., Goldstine, H.H., von Neumann, J., 1946. Preliminary discussion of the logical design of an electronic computing instrument. In: Aspray, W., Burks, A. (Eds.), Papers of John von Neumann. Report to the U.S. Army Ordnance Department, p. 1; also appears. MIT Press, Cambridge, Mass, pp. 97–146. Tomash Publishers, Los Angeles, 1987.
- Cody, W.J., Coonen, J.T., Gay, D.M., Hanson, K., Hough, D., Kahan, W., Karpinski, R., Palmer, J., Ris, F.N., Stevenson, D., 1984. A proposed radix- and word-length-independent standard for floating-point arithmetic. IEEE Micro 4 (4), 86–100. *Contains a draft of the 854 standard, which is more general than 754. The significance of this article is that it contains commentary on the standard, most of which is equally relevant to 754. However, be aware that there are some differences between this draft and the final standard.*
- Coonen, J., 1984. Contributions to a proposed standard for binary floating point arithmetic. Ph.D. thesis. University of California–Berkeley. *The only detailed discussion of how rounding modes can be used to implement efficient binary decimal conversion.*
- Darley, H.M., et al., 1989. Floating point/integer processor with divide and square root functions. U.S. Patent. 4 (878,190) October 31, 1989. *Pretty readable as patents go. Gives a high-level view of the TI 8847 chip, but doesn't have all the details of the division algorithm.*
- Demmel, J.W., Li, X., 1994. Faster numerical algorithms via exception handling. IEEE Trans. on Computers 43 (8), 983–992. *A good discussion of how the features unique to IEEE floating point can improve the performance of an important software library.*
- Freiman, C.V., 1961. Statistical analysis of certain binary division algorithms. Proc. IRE 49 (1), 91–103. *Contains an analysis of the performance of shifting-over-zeros SRT division algorithm.*
- Goldberg, D., 1991. What every computer scientist should know about floating-point arithmetic. Computing Surveys 23 (1), 5–48. *Contains an in-depth tutorial on the IEEE standard from the software point of view.*
- Goldberg, I.B., 1967. 27 bits are not enough for 8-digit accuracy. Comm. ACM 10 (2), 105–106. *This paper proposes using hidden bits and gradual underflow.*
- Gosling, J.B., 1980. Design of Arithmetic Units for Digital Computers. Springer-Verlag, New York. *A concise, well-written book, although it focuses on MSI designs.*
- Hamacher, V.C., Vranesic, Z.G., Zaky, S.G., 1984. Computer Organization, 2nd ed. McGraw-Hill, New York. *Introductory computer architecture book with a good chapter on computer arithmetic.*
- Hwang, K., 1979. Computer Arithmetic: Principles, Architecture, and Design. Wiley, New York. *This book contains the widest range of topics of the computer arithmetic books.*
- IEEE, 1985. IEEE standard for binary floating-point arithmetic. SIGPLAN Notices 22 (2), 9–25. *IEEE 754 is reprinted here.*
- Kahan, W., 1968. 7094-II system support for numerical analysis. SHARE Secretarial Distribution. SSD-159. *This system had many features that were incorporated into the IEEE floating-point standard.*
- Kahaner, D.K., 1988. Benchmarks for ‘real’ programs. SIAM News.(November).. *The benchmark presented in this article turns out to cause many underflows.*
- Knuth, D., 1981. 2nd ed. The Art of Computer Programming. Vol. II. Addison-Wesley, Reading, Mass. *Has a section on the distribution of floating-point numbers.*
- Kogge, P., 1981. The Architecture of Pipelined Computers. McGraw-Hill, New York. *Has a brief discussion of pipelined multipliers.*
- Kohn, L., Fu, S.-W., 1989. A 1,000,000 transistor microprocessor. In: IEEE Int'l. Solid-State Circuits Conf. Digest of Technical Papers, pp. 54–55. *There are several articles about the i860, but this one contains the most details about its floating-point algorithms.*
- Koren, I., 1989. Computer Arithmetic Algorithms. Prentice Hall, Englewood Cliffs, N.J..
- Leighton, F.T., 1992. Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes, Morgan Kaufmann, San Francisco. *This is an excellent book, with emphasis on the complexity analysis of algorithms. Section 1.2.1 has a nice discussion of carry-lookahead addition on a tree.*

- Magenheimer, D.J., Peters, L., Pettis, K.W., Zuras, D., 1988. Integer multiplication and division on the HP Precision architecture. *IEEE Trans. on Computers* 37 (8), 980–990. *Gives rationale for the integer- and divide-step instructions in the Precision architecture.*
- Markstein, P.W., 1990. Computation of elementary functions on the IBM RISC System/6000 processor. *IBM J. of Research and Development* 34 (1), 111–119. *Explains how to use fused multiply-add to compute correctly rounded division and square root.*
- Mead, C., Conway, L., 1980. Introduction to VLSI Systems. Addison-Wesley, Reading, Mass.
- Montoye, R.K., Hokenek, E., Runyon, S.L., 1990. Design of the IBM RISC System/6000 floating-point execution. *IBM J. of Research and Development* 34 (1), 59–70. *Describes one implementation of fused multiply-add.*
- Ngai, T.-F., Irwin, M.J., 1985. Regular, area-time efficient carry-lookahead adders. In: Proc. Seventh IEEE Symposium on Computer Arithmetic, pp. 9–15. *Describes a CLA like that of Figure J.17, where the bits flow up and then come back down.*
- Patterson, D.A., Hennessy, J.L., 2009. Computer Organization and Design: The Hardware/Software Interface, 4th Edition Morgan Kaufmann, San Francisco. *Chapter 3 is a gentler introduction to the first third of this appendix.*
- Peng, V., Samudrala, S., Gavrielov, M., 1987. On the implementation of shifters, multipliers, and dividers in VLSI floating point units. In: Proc. Eighth IEEE Symposium on Computer Arithmetic, pp. 95–102. *Highly recommended survey of different techniques actually used in VLSI designs.*
- Rowen, C., Johnson, M., Ries, P., 1988. The MIPS R3010 floating-point coprocessor. *IEEE Micro* 53–62 (June).
- Santoro, M.R., Bewick, G., Horowitz, M.A., 1989. Rounding algorithms for IEEE multipliers. In: Proc. Ninth IEEE Symposium on Computer Arithmetic, pp. 176–183. *A very readable discussion of how to efficiently implement rounding for floating-point multiplication.*
- Scott, N.R., 1985. Computer Number Systems and Arithmetic. Prentice Hall, Englewood Cliffs, N.J.
- Swartzlander, E. (Ed.), 1990. Computer Arithmetic. IEEE Computer Society Press, Los Alamitos, Calif. *A collection of historical papers in two volumes.*
- Takagi, N., Yasuura, H., Yajima, S., 1985. High-speed VLSI multiplication algorithm with a redundant binary addition tree. *IEEE Trans. on Computers* C-34 (9), 789–796. *A discussion of the binary tree signed multiplier that was the basis for the design used in the TI 8847.*
- Taylor, G.S., 1981. Compatible hardware for division and square root. In: Proc. Fifth IEEE Symposium on Computer Arithmetic, May 18–19, 1981. Ann Arbor, Mich, pp. 127–134. *Good discussion of a radix-4 SRT division algorithm.*
- Taylor, G.S., 1985. Radix 16 SRT dividers with overlapped quotient selection stages. In: Proc. Seventh IEEE Symposium on Computer Arithmetic, June 4–6, 1985, pp. 64–71 Urbana, Ill.. *Describes a very sophisticated high-radix division algorithm.*
- Weste, N., Eshraghian, K., 1993. Principles of CMOS VLSI Design: A Systems Perspective, 2nd ed. Addison-Wesley, Reading, Mass. *This textbook has a section on the layouts of various kinds of adders.*
- Williams, T.E., Horowitz, M., Alverson, R.L., Yang, T.S., 1987. A self-timed chip for division. In: Advanced Research in VLSI, Proc. 1987 Stanford Conf. MIT Press, Cambridge, Mass. *Describes a divider that tries to get the speed of a combinational design without using the area that would be required by one.*

## Exercises

- J.1 [12]<J.2> Using  $n$  bits, what is the largest and smallest integer that can be represented in the two's complement system?
- J.2 [20/25]<J.2> In the subsection “Signed Numbers” (page J-7), it was stated that two's complement overflows when the carry into the high-order bit position is different from the carry-out from that position.

- a. [20]<J.2> Give examples of pairs of integers for all four combinations of carry-in and carry-out. Verify the rule stated above.
- b. [25]<J.2> Explain why the rule is always true.
- J.3 [12]<J.2> Using 4-bit binary numbers, multiply  $-8 \times -8$  using Booth recoding.
- J.4 [15]<J.2> Equations J.2.1 and J.2.2 are for adding two  $n$ -bit numbers. Derive similar equations for subtraction, where there will be a borrow instead of a carry.
- J.5 [25]<J.2> On a machine that doesn't detect integer overflow in hardware, show how you would detect overflow on a signed addition operation in software.
- J.6 [15/15/20]<J.3> Represent the following numbers as single-precision and double-precision IEEE floating-point numbers:
  - a. [15]<J.3> 10.
  - b. [15]<J.3> 10.5.
  - c. [20]<J.3> 0.1.
- J.7 [12/12/12/12/12]<J.3> Below is a list of floating-point numbers. In single precision, write down each number in binary, in decimal, and give its representation in IEEE arithmetic.
  - a. [12]<J.3> The largest number less than 1.
  - b. [12]<J.3> The largest number.
  - c. [12]<J.3> The smallest positive normalized number.
  - d. [12]<J.3> The largest denormal number.
  - e. [12]<J.3> The smallest positive number.
- J.8 [15]<J.3> Is the ordering of nonnegative floating-point numbers the same as integers when denormalized numbers are also considered?
- J.9 [20]<J.3> Write a program that prints out the bit patterns used to represent floating-point numbers on your favorite computer. What bit pattern is used for NaN?
- J.10 [15]<J.4> Using  $p=4$ , show how the binary floating-point multiply algorithm computes the product of  $1.875 \times 1.875$ .
- J.11 [12/10]<J.4> Concerning the addition of exponents in floating-point multiply:
  - a. [12]<J.4> What would the hardware that implements the addition of exponents look like?
  - b. [10]<J.4> If the bias in single precision were 129 instead of 127, would addition be harder or easier to implement?
- J.12 [15/12]<J.4> In the discussion of overflow detection for floating-point multiplication, it was stated that (for single precision) you can detect an overflowed exponent by performing exponent addition in a 9-bit adder.

- a. [15]<J.4> Give the exact rule for detecting overflow.
  - b. [12]<J.4> Would overflow detection be any easier if you used a 10-bit adder instead?
- J.13 [15/10]<J.4> Floating-point multiplication:
- a. [15]<J.4> Construct two single-precision floating-point numbers whose product doesn't overflow until the final rounding step.
  - b. [10]<J.4> Is there any rounding mode where this phenomenon cannot occur?
- J.14 [15]<J.4> Give an example of a product with a denormal operand but a normalized output. How large was the final shifting step? What is the maximum possible shift that can occur when the inputs are double-precision numbers?
- J.15 [15]<J.5> Use the floating-point addition algorithm on page J-23 to compute  $1.010_2 - .1001_2$  (in 4-bit precision).
- J.16 [10/15/20/20/20]<J.5> In certain situations, you can be sure that  $a+b$  is exactly representable as a floating-point number, that is, no rounding is necessary.
- a. [10]<J.5> If  $a, b$  have the same exponent and different signs, explain why  $a+b$  is exact. This was used in the subsection "Speeding Up Addition" on page J-25.
  - b. [15]<J.5> Give an example where the exponents differ by 1,  $a$  and  $b$  have different signs, and  $a+b$  is not exact.
  - c. [20]<J.5> If  $a \geq b \geq 0$ , and the top two bits of  $a$  cancel when computing  $a-b$ , explain why the result is exact (this fact is mentioned on page J-22).
  - d. [20]<J.5> If  $a \geq b \geq 0$ , and the exponents differ by 1, show that  $a-b$  is exact unless the high order bit of  $a-b$  is in the same position as that of  $a$  (mentioned in "Speeding Up Addition," page J-25).
  - e. [20]<J.5> If the result of  $a-b$  or  $a+b$  is denormal, show that the result is exact (mentioned in the subsection "Underflow," on page J-36).
- J.17 [15/20]<J.5> Fast floating-point addition (using parallel adders) for  $p=5$ .
- a. [15]<J.5> Step through the fast addition algorithm for  $a+b$ , where  $a = 1.0111_2$  and  $b = .11011_2$ .
  - b. [20]<J.5> Suppose the rounding mode is toward  $+\infty$ . What complication arises in the above example for the adder that assumes a carry-out? Suggest a solution.
- J.18 [12]<J.4, J.5> How would you use two parallel adders to avoid the final round-up addition in floating-point multiplication?
- J.19 [30/10]<J.5> This problem presents a way to reduce the number of addition steps in floating-point addition from three to two using only a single adder.
- a. [30]<J.5> Let  $A$  and  $B$  be integers of opposite signs, with  $a$  and  $b$  their magnitudes. Show that the following rules for manipulating the unsigned numbers  $a$  and  $b$  gives  $A+B$ .

1. Complement one of the operands.
  2. Use end-around carry to add the complemented operand and the other (uncomplemented) one.
  3. If there was a carry-out, the sign of the result is the sign associated with the uncomplemented operand.
  4. Otherwise, if there was no carry-out, complement the result, and give it the sign of the complemented operand.
- b. [10] <J.5> Use the above to show how steps 2 and 4 in the floating-point addition algorithm on page J-23 can be performed using only a single addition.
- J.20 [20/15/20/15/20/15] <J.6> Iterative square root.
- a. [20] <J.6> Use Newton's method to derive an iterative algorithm for square root. The formula will involve a division.
  - b. [15] <J.6> What is the fastest way you can think of to divide a floating-point number by 2?
  - c. [20] <J.6> If division is slow, then the iterative square root routine will also be slow. Use Newton's method on  $f(x) = 1/x^2 - a$  to derive a method that doesn't use any divisions.
  - d. [15] <J.6> Assume that the ratio division by 2 : floating-point add : floating-point multiply is 1:2:4. What ratios of multiplication time to divide time makes each iteration step in the method of part (c) faster than each iteration in the method of part (a)?
  - e. [20] <J.6> When using the method of part (a), how many bits need to be in the initial guess in order to get double-precision accuracy after three iterations? (You may ignore rounding error.)
  - f. [15] <J.6> Suppose that when spice runs on the TI 8847, it spends 16.7% of its time in the square root routine (this percentage has been measured on other machines). Using the values in Figure J.36 and assuming three iterations, how much slower would spice run if square root were implemented in software using the method of part(a)?
- J.21 [10/20/15/15/15] <J.6> Correctly rounded iterative division. Let  $a$  and  $b$  be floating-point numbers with  $p$ -bit significands ( $p = 53$  in double precision). Let  $q$  be the exact quotient  $q = a/b$ ,  $1 \leq q < 2$ . Suppose that  $\bar{q}$  is the result of an iteration process, that  $\bar{q}$  has a few extra bits of precision, and that  $0 < q - \bar{q} < 2^{-p}$ . For the following, it is important that  $\bar{q} < q$ , even when  $q$  can be exactly represented as a floating-point number.
- a. [10] <J.6> If  $x$  is a floating-point number, and  $1 \leq x < 2$ , what is the next representable number after  $x$ ?
  - b. [20] <J.6> Show how to compute  $q'$  from  $\bar{q}$ , where  $q'$  has  $p+1$  bits of precision and  $|q - q'| < 2^{-p}$ .
  - c. [15] <J.6> Assuming round to nearest, show that the correctly rounded quotient is either  $q'$ ,  $q' - 2^{-p}$ , or  $q' + 2^{-p}$ .

- d. [15]<J.6> Give rules for computing the correctly rounded quotient from  $q'$  based on the low-order bit of  $q'$  and the sign of  $a - bq'$ .
- e. [15]<J.6> Solve part (c) for the other three rounding modes.
- J.22 [15]<J.6> Verify the formula on page J-30. (*Hint:* If  $x_n = x_0(2 - x_0b) \times \prod_{i=1,n} [1 + (1 - x_0b)^{2^i}]$ , then  $2 - x_n b = 2 - x_0 b (2 - x_0 b) \prod [1 + (1 - x_0 b)^{2^i}] = 2 - [1 - (1 - x_0 b)^2] \prod [1 + (1 - x_0 b)^{2^i}]$ .)
- J.23 [15]<J.7> Our example that showed that double rounding can give a different answer from rounding once used the round-to-even rule. If halfway cases are always rounded up, is double rounding still dangerous?
- J.24 [10/10/20/20]<J.7> Some of the cases of the italicized statement in the “Precisions” subsection (page J-33) aren’t hard to demonstrate.
- [10]<J.7> What form must a binary number have if rounding to  $q$  bits followed by rounding to  $p$  bits gives a different answer than rounding directly to  $p$  bits?
  - [10]<J.7> Show that for multiplication of  $p$ -bit numbers, rounding to  $q$  bits followed by rounding to  $p$  bits is the same as rounding immediately to  $p$  bits if  $q \geq 2p$ .
  - [20]<J.7> If  $a$  and  $b$  are  $p$ -bit numbers with the same sign, show that rounding  $a+b$  to  $q$  bits followed by rounding to  $p$  bits is the same as rounding immediately to  $p$  bits if  $q \geq 2p+1$ .
  - [20]<J.7> Do part (c) when  $a$  and  $b$  have opposite signs.
- J.25 [Discussion]<J.7> In the MIPS approach to exception handling, you need a test for determining whether two floating-point operands could cause an exception. This should be fast and also not have too many false positives. Can you come up with a practical test? The performance cost of your design will depend on the distribution of floating-point numbers. This is discussed in Knuth [1981] and the Hamming paper in Swartzlander [1990].
- J.26 [12/12/10]<J.8> Carry-skip adders.
- [12]<J.8> Assuming that time is proportional to logic levels, how long does it take an  $n$ -bit adder divided into (fixed) blocks of length  $k$  bits to perform an addition?
  - [12]<J.8> What value of  $k$  gives the fastest adder?
  - [10]<J.8> Explain why the carry-skip adder takes time  $O(\sqrt{n})$ .
- J.27 [10/15/20]<J.8> Complete the details of the block diagrams for the following adders.
- [10]<J.8> In Figure J.15, show how to implement the “1” and “2” boxes in terms of AND and OR gates.
  - [15]<J.8> In Figure J.19, what signals need to flow from the adder cells in the top row into the “C” cells? Write the logic equations for the “C” box.
  - [20]<J.8> Show how to extend the block diagram in J.17 so it will produce the carry-out bit  $c_8$ .

- J.28 [15]<J.9>For ordinary Booth recoding, the multiple of  $b$  used in the  $i$ th step is simply  $a_{i-1} - a_i$ . Can you find a similar formula for radix-4 Booth recoding (overlapped triplets)?
- J.29 [20]<J.9>Expand Figure J.29 in the fashion of J.27, showing the individual adders.
- J.30 [25]<J.9>Write out the analog of Figure J.25 for radix-8 Booth recoding.
- J.31 [18]<J.9>Suppose that  $a_{n-1} \dots a_1 a_0$  and  $b_{n-1} \dots b_1 b_0$  are being added in a signed-digit adder as illustrated in the example on page J-53. Write a formula for the  $i$ th bit of the sum,  $s_i$ , in terms of  $a_i, a_{i-1}, a_{i-2}, b_i, b_{i-1}$ , and  $b_{i-2}$ .
- J.32 [15]<J.9>The text discussed radix-4 SRT division with quotient digits of  $-2, -1, 0, 1, 2$ . Suppose that  $3$  and  $-3$  are also allowed as quotient digits. What relation replaces  $|r_i| \leq 2b/3$ ?
- J.33 [25/20/30]<J.9>Concerning the SRT division table, Figure J.34:
- [25]<J.9>Write a program to generate the results of Figure J.34.
  - [20]<J.9>Note that Figure J.34 has a certain symmetry with respect to positive and negative values of  $P$ . Can you find a way to exploit the symmetry and only store the values for positive  $P$ ?
  - [30]<J.9>Suppose a carry-save adder is used instead of a propagate adder. The input to the quotient lookup table will be  $k$  bits of divisor and  $l$  bits of remainder, where the remainder bits are computed by summing the top  $l$  bits of the sum and carry registers. What are  $k$  and  $l$ ? Write a program to generate the analog of Figure J.34.
- J.34 [12/12/12]<J.9, J.12>The first several million Pentium chips produced had a flaw that caused division to sometimes return the wrong result. The Pentium uses a radix-4 SRT algorithm similar to the one illustrated in the example on page J-56 (but with the remainder stored in carry-save format; see Exercise J.33(c)). According to Intel, the bug was due to five incorrect entries in the quotient lookup table.
- [12]<J.9, J.12>The bad entries should have had a quotient of plus or minus 2, but instead had a quotient of 0. Because of redundancy, it's conceivable that the algorithm could "recover" from a bad quotient digit on later iterations. Show that this is not possible for the Pentium flaw.
  - [12]<J.9, J.12>Since the operation is a floating-point divide rather than an integer divide, the SRT division algorithm on page J-45 must be modified in two ways. First, step 1 is no longer needed, since the divisor is already normalized. Second, the very first remainder may not satisfy the proper bound ( $|r| \leq 2b/3$  for Pentium; see page J-55). Show that skipping the very first left shift in step 2(a) of the SRT algorithm will solve this problem.
  - [12]<J.9, J.12>If the faulty table entries were indexed by a remainder that could occur at the very first divide step (when the remainder is the divisor), random testing would quickly reveal the bug. This didn't happen. What does that tell you about the remainder values that index the faulty entries?

- J.35 [12]<J.6, J.9>The discussion of the remainder-step instruction assumed that division was done using a bit-at-a-time algorithm. What would have to change if division were implemented using a higher-radix method?
- J.36 [25]<J.9>In the array of Figure J.28, the fact that an array can be pipelined is not exploited. Can you come up with a design that feeds the output of the bottom CSA into the bottom CSAs instead of the top one, and that will run faster than the arrangement of Figure J.28?

---

|     |                                                                               |      |
|-----|-------------------------------------------------------------------------------|------|
| K.1 | Introduction                                                                  | K-2  |
| K.2 | A Survey of RISC Architectures for Desktop, Server, and Embedded<br>Computers | K-3  |
| K.3 | The Intel 80x86                                                               | K-30 |
| K.4 | The VAX Architecture                                                          | K-50 |
| K.5 | The IBM 360/370 Architecture for Mainframe Computers                          | K-69 |
| K.6 | Historical Perspective and References                                         | K-75 |

# K

---

## Survey of Instruction Set Architectures

RISC: any computer announced after 1985.

Steven Przybylski  
*A Designer of the Stanford MIPS*

## K.1 Introduction

This appendix covers 10 instruction set architectures, some of which remain a vital part of the IT industry and some of which have retired to greener pastures. We keep them all in part to show the changes in fashion of instruction set architecture over time.

We start with eight RISC architectures, using RISC V as our basis for comparison. There are billions of dollars of computers shipped each year for ARM (including Thumb-2), MIPS (including microMIPS), Power, and SPARC. ARM dominates in both the PMD (including both smart phones and tablets) and the embedded markets.

The 80x86 remains the highest dollar-volume ISA, dominating the desktop and the much of the server market. The 80x86 did not get traction in either the embedded or PMD markets, and has started to lose ground in the server market. It has been extended more than any other ISA in this book, and there are no plans to stop it soon. Now that it has made the transition to 64-bit addressing, we expect this architecture to be around, although it may play a smaller role in the future than it did in the past 30 years.

The VAX typifies an ISA where the emphasis was on code size and offering a higher level machine language in the hopes of being a better match to programming languages. The architects clearly expected it to be implemented with large amounts of microcode, which made single chip and pipelined implementations more challenging. Its successor was the Alpha, a RISC architecture similar to MIPS and RISC V, but which had a short life.

The vulnerable IBM 360/370 remains a classic that set the standard for many instruction sets to follow. Among the decisions the architects made in the early 1960s were:

- 8-bit byte
- Byte addressing
- 32-bit words
- 32-bit single precision floating-point format + 64-bit double precision floating-point format
- 32-bit general-purpose registers, separate 64-bit floating-point registers
- Binary compatibility across a family of computers with different cost-performance
- Separation of architecture from implementation

As mentioned in Chapter 2, the IBM 370 was extended to be virtualizable, so it had the lowest overhead for a virtual machine of any ISA. The IBM 360/370 remains the foundation of the IBM mainframe business in a version that has extended to 64 bits.

**K.2****A Survey of RISC Architectures for Desktop, Server, and Embedded Computers****Introduction**

We cover two groups of Reduced Instruction Set Computer (RISC) architectures in this section. The first group is the desktop, server RISCs, and PMD processors:

- Advanced RISC Machines ARMv8, AArch64, the 64-bit ISA,
- MIPS64, version 6, the most recent the 64-bit ISA,
- Power version 3.0, which merges the earlier IBM Power architecture and the PowerPC architecture.
- RISC-V, specifically RV64G, the 64-bit extension of RISC-V.
- SPARCv9, the 64-bit ISA.

As Figure K.1 shows these architectures are remarkably similar.

There are two other important historical RISC processors that are almost identical to those in the list above: the DEC Alpha processor, which was made by Digital Equipment Corporation from 1992 to 2004 and is almost identical to MIPS64. Hewlett-Packard's PA-RISC was produced by HP from about 1986 to 2005, when it was replaced by Itanium. PA-RISC is most closely related to the Power ISA, which emerged from the IBM Power design, itself a descendant of IBM 801.

The second group is the embedded RISCs designed for lower-end applications:

- Advanced RISC Machines, Thumb-2: an 32-bit instruction set with 16-bit and 32-bit instructions. The architecture includes features from both ARMv7 and ARMv8.
- microMIPS64: a version of the MIPS64 instruction set with 16-bit instructions, and
- RISC-V Compressed extension (RV64GC), a set of 16-bit instructions added to RV64G

Both RV64GC and microMIPS64 have corresponding 32-bit versions: RV32GC and microMIPS32.

Since the comparison of the base 32-bit or 64-bit desktop and server architecture will examine the differences among those ISAs, our discussion of the embedded architectures focuses on the 16-bit instructions. Figure K.2 shows that these embedded architectures are also similar. In all three, the 16-bit instructions are versions of 32-bit instructions, typically with a restricted set of registers. The idea is to reduce the code size by replacing common 32-bit instructions with 16-bit versions. For RV32GC or Thumb-2, including the 16-bit instructions yields a reduction in code size to about 0.73 of the code size using only the 32-bit ISA (either RV32G or ARMv7).

|                                         | <b>ARMv8</b>                             | <b>MIPS64 R6</b>                      | <b>Power v3.0</b>                     | <b>RV64G</b>                          | <b>SPARCv9</b>                        |
|-----------------------------------------|------------------------------------------|---------------------------------------|---------------------------------------|---------------------------------------|---------------------------------------|
| Original date (base ISA)                | 1986                                     | 1986                                  | 1990                                  | 2016                                  | 1987                                  |
| Date of this ISA                        | 2011                                     | 2014                                  | 2013                                  | 2016                                  | 2008                                  |
| Instruction size (bits)                 | 32                                       | 32                                    | 32                                    | 32                                    | 32                                    |
| Address space (size, model)             | 64 bits (flat)                           | 64 bits, flat                         | 64 bits, flat                         | 64 bits, flat                         | 64 bits, flat                         |
| Data alignment                          | Aligned preferred                        | Aligned preferred                     | Unaligned                             | Aligned preferred                     | Aligned                               |
| Data addressing modes                   | 8 (including scaled, pre/post increment) | 1 (+1 for FP only)                    | 4                                     | 1                                     | 2                                     |
| Integer registers (number, model, size) | 31 GPR $\times$ 64, plus stack pointer   | 31 GPR $\times$ 64 bits               |
| Separate floating-point registers       | 32x32 or 32x64 bits                      | 32 $\times$ 32 or 32 $\times$ 64 bits | 32 $\times$ 32 or 32 $\times$ 64 bits | 32 $\times$ 32 or 32 $\times$ 64 bits | 32 $\times$ 32 or 32 $\times$ 64 bits |
| Floating-point format                   | IEEE 754 single, double                  | IEEE 754 single, double               | IEEE 754 single, double               | IEEE 754 single, double               | IEEE 754 single, double               |

**Figure K.1 Summary of the most recent version of five architectures for desktop, server, and PMD use (all had earlier versions).** Except for the number of data address modes and some instruction set details, the integer instruction sets of these architectures are very similar. Contrast this with Figure K.29. In ARMv8, register 31 is a 0 (like register 0 in the other architectures), but when it is used in a load or store, it is the current stack pointer, a special purpose register. We can either think of SP-based addressing as a different mode (which is how the assembly mnemonics operate) or as simply a register + offset addressing mode (which is how the instruction is encoded).

|                                                                                        | <b>microMIPS64</b>                                      | <b>RV64GC</b>             | <b>Thumb-2</b>              |
|----------------------------------------------------------------------------------------|---------------------------------------------------------|---------------------------|-----------------------------|
| Date announced                                                                         | 2009                                                    | 2016                      | 2003                        |
| Instruction size (bits)                                                                | 16/32                                                   | 16/32                     | 16/32                       |
| Address space (size, model)                                                            | 32/64 bits, flat                                        | 32/64 bits flat           | 32 bits, flat               |
| Data alignment                                                                         | Aligned                                                 | Aligned, preferred        | Aligned                     |
| Data addressing modes                                                                  | 2                                                       | 1                         | 6                           |
| Integer registers (number, model, size)                                                | 31 GPR $\times$ 64 bits                                 | 31 GPR $\times$ 64 bits   | 15 GPR $\times$ 32 bits     |
| Integer registers accessible by most 16-bit instructions (which use should specifiers) | 8 GPR + SP + GP +RA<br>GPRs: 0, 2-7, 17, or 2-7, 16, 17 | 8 GPRs + SP<br>GPRs: 8-15 | 8 GPR + SP $\times$ 32 bits |

**Figure K.2 Summary of three recent architectures for embedded applications.** All three use 16-bit extensions of a base instruction set. Except for number of data address modes and a number of instruction set details, the integer instruction sets of these architectures are similar. Contrast this with Figure K.29. An earlier 16-bit version of the MIPS instruction set, called MIPS16, was created in 1995 and was replaced by microMIPS32 and microMIPS64. The first Thumb architecture had only 16-bit instructions and was created in 1996. Thumb-2 is built primarily on ARMv7, the 32-bit ARM instruction set; it offers 16 registers. RISC-V also defines RV32E, which has only 16 registers, includes the 16-bit instructions, and cannot have floating point. It appears that most implementations for embedded applications opt for RV32C or RV64GC.

A key difference among these three architectures is the structure of the base 32-bit ISA. In the case of RV64GC, the 32-bit instructions are exactly those of RV64G. This is possible because RISC V planned for the 16-bit option from the beginning, and branch addresses and jump addresses are specified to 16-bit boundaries. In the case of microMIPS64, the base ISA is MIPS64, with one change: branch and jump offsets are interpreted as 16-bit rather than 32-bit aligned. (microMIPS also uses the encoding space that was reserved in MIPS64 for user-defined instruction set extensions; such extensions are not part of the base ISA.)

Thumb-2 uses a slightly different approach. The 32-bit instructions in Thumb-2 are mostly a subset of those in ARMv7; certain features that were dropped in ARMv8 are not included (e.g., conditional execution of most instructions and the ability to write the PC as a GPR). Thumb-2 also includes a few dozen instructions introduced in ARMv8, specifically bit field manipulation, additional system instructions, and synchronization support. Thus, the 32-bit instructions in Thumb-2 constitute a unique ISA.

Earlier versions of the 16-bit instruction sets for MIPS (MIPS16) and ARM (Thumb), took the approach of creating a separate mode, invoked by a procedure call, to transfer control to a code segment that employed only 16-bit instructions.

The 16-bit instruction set was not complete and was only intended for user programs that were code-size critical.

One complication of this description is that some of the older RISCs have been extended over the years. We decided to describe the most recent versions of the architectures: ARMv8 (the 64-bit architecture AArch64), MIPS64 R6, Power v3.0, RV64G, and SPARC v9 for the desktop/server/PMD, and the 16-bit subset of the ISAs for microMIPS64, RV64GC, and Thumb-2.

The remaining sections proceed as follows. After discussing the addressing modes and instruction formats of our RISC architectures, we present the survey of the instructions in five steps:

- Instructions found in the RV64G core, described in Appendix A.
- Instructions not found in the RV64G or RV64GC but found in two or more of the other architectures. We describe and organize these by functionality, e.g. instructions that support extended integer arithmetic.
- Instruction groups unique to ARM, MIPS, Power, or SPARC, organized by function.
- Multimedia extensions of the desktop/server/PMD RISCs
- Digital signal-processing extensions of the embedded RISCs

Although the majority of the instructions in these architectures are included, we have not included every single instruction; this is especially true for the Power and ARM ISAs, which have *many* instructions.

## Addressing Modes and Instruction Formats

Figure K.3 shows the data addressing modes supported by the desktop/server/PMD architectures. Since all, but ARM, have one register that always has the value 0 when used in address modes, the absolute address mode with limited range can be synthesized using register 0 as the base in displacement addressing. (This register can be changed by arithmetic-logical unit (ALU) operations in PowerPC, but is always zero when it is used in an address calculation.) Similarly, register indirect addressing is synthesized by using displacement addressing with an offset of 0. Simplified addressing modes is one distinguishing feature of RISC architectures.

As Figure K.4 shows, the embedded architectures restrict the registers that can be accessed with the 16-bit instructions, typically to only 8 registers, for most instructions, and a few special instructions that refer to other registers. Figure K.5 shows the data addressing modes supported by the embedded architectures in their 16-bit instruction mode. These versions of load/store instructions restrict the registers that can be used in address calculations, as well as significantly shorten the immediate fields, used for displacements.

References to code are normally PC-relative, although jump register indirect is supported for returning from procedures, for case statements, and for pointer function calls. One variation is that PC-relative branch addresses are often shifted left 2 bits before being added to the PC for the desktop RISCs, thereby increasing the branch distance. This works because the length of all instructions for the desktop

|                                                                                  | <b>ARMv8</b> | <b>MIPS64 R6</b> | <b>Power v3.0</b> | <b>RV64G</b> | <b>SPARCv9</b> |
|----------------------------------------------------------------------------------|--------------|------------------|-------------------|--------------|----------------|
| Register + offset (displacement or based)                                        | B, H, W, D   | B, H, W, D       | B, H, W, D        | B, H, W, D   | B, H, W, D     |
| Register + register (indexed)                                                    | B, H, W, D   |                  | B, H, W, D        |              | B, H, W, D     |
| Register + scaled register (scaled)                                              | B, H, W, D   | W,D              |                   |              |                |
| Register + register + offset                                                     | B, H, W, D   |                  |                   |              |                |
| Register + offset & update register to effective address (based with update)     | B, H, W, D   |                  | B, H, W, D        |              |                |
| Register & update register to register + offset (register with update)           | B, H, W, D   |                  |                   |              |                |
| Register + Register & update register to effective address (indexed with update) | B, H, W, D   |                  | B, H, W, D        |              |                |
| PC-relative (PC + displacement)                                                  | W, D         | W, D             |                   |              |                |

**Figure K.3** Summary of data addressing modes supported by the desktop architectures, where B, H, W, D indicate what datatypes can use the addressing mode. Note that ARM includes two different types of address modes with updates, one of which is included in Power.

| Register specifier      | microMIPS64                                   | RV64GC                      | Thumb-2                     |
|-------------------------|-----------------------------------------------|-----------------------------|-----------------------------|
| 3-bit                   | 2-7,16, 17                                    | 8-15                        | 0-7                         |
| stack pointer register  | 29                                            | 2                           | 0 (when used in load/store) |
| global pointer register | 28                                            |                             |                             |
| return address register | 31                                            | 1                           | 14                          |
| Using special register  | stack pointer or global pointer; 5-bit offset | stack pointer; 5-bit offset | stack pointer; 8-bit offset |

**Figure K.4** Register encodings for the 16-bit subsets of microMIPS64, RV64GC, and Thumb-2, including the core general purpose registers, and special-purpose registers accessible by some instructions.

| Addressing mode                           | microMIPS64                                   | RV64GC                           | Thumb-2                          |
|-------------------------------------------|-----------------------------------------------|----------------------------------|----------------------------------|
| Register + offset (displacement or based) | 4-bit offset, one of 8 registers              | 5-bit offset, one of 8 registers | 5-bit offset, one of 8 registers |
| PC-relative data                          |                                               |                                  | word only; 8-bit offset          |
| Using special register                    | stack pointer or global pointer; 5-bit offset | stack pointer; 5-bit offset      | stack pointer; 8-bit offset      |

**Figure K.5** Summary of data addressing modes supported by the embedded architectures. microMIPS64, RV64c, and Thumb-2 show only the modes supported in 16-bit instruction formats. The stack pointer in RV64GC and micro-MIPS64 is a designed GPR; it is another version of r31 is Thumb-2. In microMIPS64, the global pointer is register 30 and is used by the linkage convention to point to the global variable data pool. Notice that typically only 8 registers are accessible as base registers (and as we will see as ALU sources and destinations).

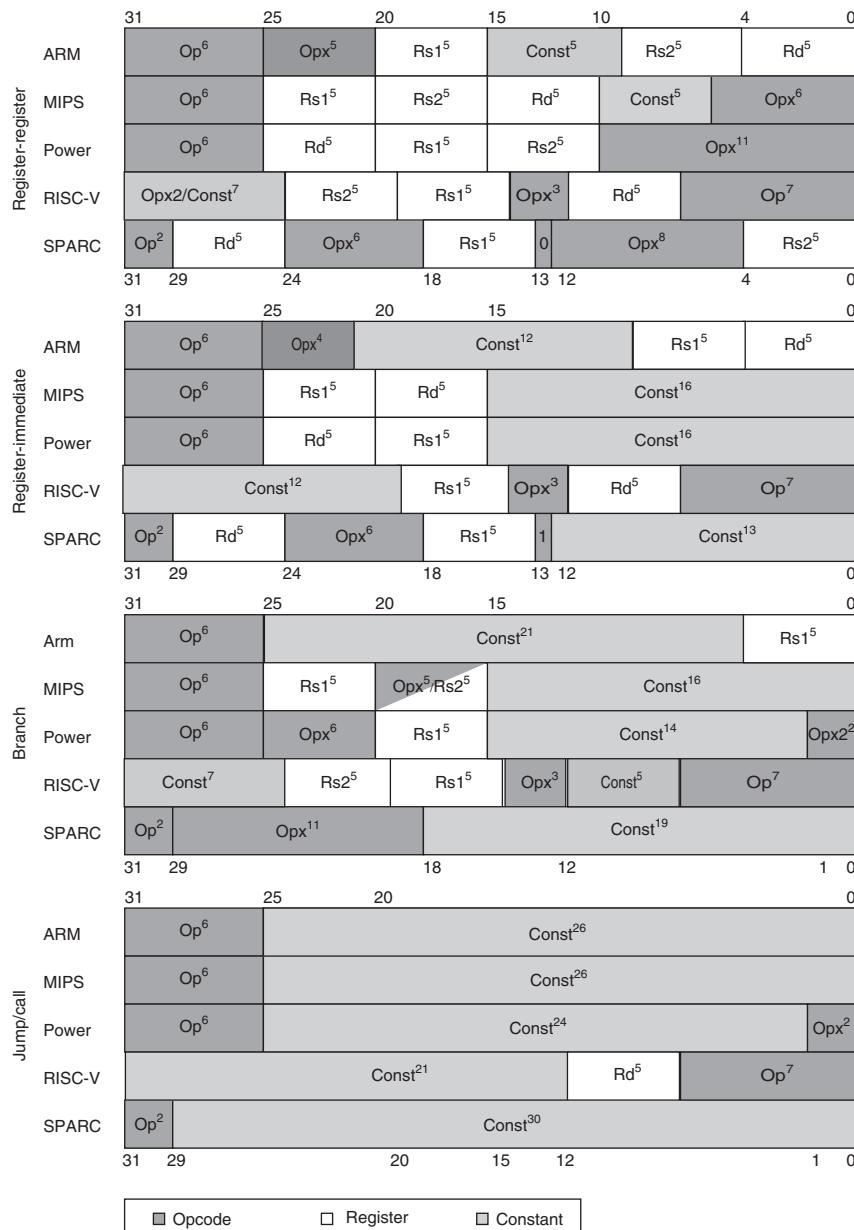
RISCs is 32 bits and instructions must be aligned on 32-bit words in memory. Embedded architectures and RISC V (when extended) have 16-bit-long instructions and usually shift the PC-relative address by 1 for similar reasons.

Figure K.6 shows the most important instruction formats of the desktop/server/PMD RISC instructions. Each instruction set architecture uses four primary instruction formats, which typically include 90–98% of the instructions. The register-register format is used for register-register ALU instructions, while the ALU immediate format is used for ALU instructions with an immediate operand and also for loads and stores. The branch format is used for conditional branches, and the jump/call format for unconditional branches (jumps) and procedures calls.

There are a number of less frequently used instruction formats that Figure K.6 leaves out. Figure K.7 summarizes these for the desktop/server/PMD architectures.

Unlike, their 32-bit base architectures, the 16-bit extensions (microMIPS64, RV64GC, and Thumb-2) are focused on minimizing code. As a result, there are a larger number of instruction formats, even though there are far fewer instructions.

**K-8** ■ Appendix K *Survey of Instruction Set Architectures*



**Figure K.6 Instruction formats for desktop/server RISC architectures.** These four formats are found in all five architectures. (The superscript notation in this figure means the width of a field in bits.) Although the register fields are located in similar pieces of the instruction, be aware that the destination and two source fields are sometimes scrambled. Op = the main opcode, Opx = an opcode extension, Rd = the destination register, Rs1 = source register 1, Rs2 = source register 2, and Const = a constant (used as an immediate, address, mask, or sift amount). Although the labels on the instruction formats tell where various instructions are encoded, there are variations. For example, loads and stores, both use the ALU immediate form in MIPS. In RISC-V, loads use the ALU immediate format, while stores use the branch format.

| Architecture | Additional instruction formats                                          | Format function and use                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|--------------|-------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ARMv8        | At least 10 (many small variations); major forms are shown.             | Logical immediates with 13-bit immediate field.<br>Shifts with constant amount.(16-bit opcode)<br>16-bit immediate form<br>Exclusive operations: three register fields<br>Branch register: long opcode<br>Load/store with address mode bits.                                                                                                                                                                                                                                                                                                               |
| MIPS64       | 1                                                                       | A PC-relative set of load/stores using register-immediate format but with 18-bit immediates (since the other source is the PC).                                                                                                                                                                                                                                                                                                                                                                                                                            |
| Power        | 9 (not including a number of small variations or the vector extensions) | DQ-mode: uses the ALU immediate form but takes four bits of the displacement for other functions.<br>DS-mode: uses the ALU immediate form but takes two bits of the displacement for other functions.<br>DX-form: Like register-immediate, but with a register-source replaced by PC.<br>MD, MDS formats: like register-register but used for shifts and rotates.<br>X, XS, and several minor variations: used for indexed addressing modes, shifts, and a variety of extended purposes.<br>Z22, Z23 formats: used for manipulating floating point numbers |
| RV64         | 2                                                                       | SB format: a variant of the branch format with different immediate treatment<br>UJ format: a variant of the jump/call format with different immediate treatment                                                                                                                                                                                                                                                                                                                                                                                            |
| SPARC        | 3                                                                       | Another format for conditional branches containing 3 more bits of displacement (22 total versus 19) but no prediction hints.<br>A format with 22-bit immediate used to load the upper half of a register,<br>A format for conditional branches based on a register compare with zero.                                                                                                                                                                                                                                                                      |

**Figure K.7 Other instruction formats beyond the four major formats of the previous figure.** In some cases, there are formats very similar to one of the four core formats, but where a register field is used for other purposes. The Power architecture also includes a number of formats for vector operations.

microMIPs64 and RV64GC have eight and seven major formats, respectively, and Thumb-2 has 15. As Figure K.8 shows, these involve varying number of register operands (0 to 3), different immediate sizes, and even different size register specifiers, with a small number of registers accessible by most instructions, and fewer instructions able to access all 32 registers.

## Instructions

The similarities of each architecture allow simultaneous descriptions, starting with the operations equivalent to the RISC-V 64-bit ISA.

| Architecture | Opcode<br>main:<br>extended | Register<br>specifiers<br>length | Immediate<br>field<br>length | Typical instructions                                                                                                                                                          |
|--------------|-----------------------------|----------------------------------|------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| microMIPS64  | 6                           | none                             | 10                           | Jumps                                                                                                                                                                         |
|              | 6                           | 1x5                              | 5                            | Register-register operation (32 registers) and Load using SP as base register; any destination                                                                                |
|              | 6                           | 1x3                              | 7                            | Branches equal/not equal zero. Loads using GP. as base.                                                                                                                       |
|              | 6:4                         | 2x3                              |                              | Register-register operation, rd/rs1, and rs2; 8 registers                                                                                                                     |
|              | 6:1                         | 2x3                              | 3                            | Register-register immediate, rd/rs1, and rs2; 8 registers                                                                                                                     |
|              | 6                           | 2x3                              | 4                            | Loads and stores; 8 registers                                                                                                                                                 |
|              | 6:4                         | 2x3                              |                              | Register-register operation, rd, and rs1; 8 registers                                                                                                                         |
|              | 6                           | 2x5                              |                              | Register-register operation; 32 registers.                                                                                                                                    |
| RV64GC       | 2:3                         |                                  | 11                           | Jumps                                                                                                                                                                         |
|              | 2:3                         | 1x3                              | 7                            | Branch                                                                                                                                                                        |
|              | 2:3                         | 1x3                              | 8                            | Immediate one source register.                                                                                                                                                |
|              | 2:3                         | 1x5                              | 6                            | Store using SP as base.                                                                                                                                                       |
|              | 2:3                         | 1x5                              | 6                            | ALU immediate and load using SP as base.                                                                                                                                      |
|              | 2:4                         | 2x5                              |                              | Register-register operation                                                                                                                                                   |
|              | 2:3                         | 2x3                              | 5                            | Loads and stores using 8 registers.                                                                                                                                           |
| Thumb-2      | 3:2                         | 2x3                              | 5                            | Shift, move, load/store word/byte                                                                                                                                             |
|              | 3:2                         | 1x3                              | 8                            | immediates: add, subtract, move, and compare                                                                                                                                  |
|              | 4:1                         | 1x3                              | 8                            | Load/store with stack pointer as base, Add to SP or PC, Load/store multiple                                                                                                   |
|              | 4:3                         | 3x3                              |                              | Load register indexed                                                                                                                                                         |
|              | 4:4                         |                                  | 8                            | Conditional branch, system instruction                                                                                                                                        |
|              | 4:12                        |                                  |                              | Miscellaneous: 22 different instructions with 12 formats (includes compare and branch on zero, pop/push registers, adjust stack pointer, reverse bytes, IF-THEN instruction). |
|              | 5                           | 1x3                              | 8                            | Load relative to PC                                                                                                                                                           |
|              | 5                           |                                  | 11                           | Unconditional branch                                                                                                                                                          |
|              | 6:1                         | 3x3                              |                              | Add/subtract                                                                                                                                                                  |
|              | 6:3                         | 1x4, 1x3                         |                              | Special data processing                                                                                                                                                       |
|              | 6:4                         | 2x3                              |                              | Logical data processing                                                                                                                                                       |
|              | 6:6                         | 1x4                              |                              | Branch and change instruction set (ARM vs. Thumb)                                                                                                                             |

**Figure K.8 Instruction formats for the 16-bit instructions of microMIPS64, RV64GC, and Thumb-2.** For instructions with a destination and two sources, but only two register fields, the instruction uses one of the registers as both source and destination. Note that the extended opcode field (or function field) and immediate field sometimes overlap or are identical. For RV64GC and microMIPS64, all the formats are shown; for Thumb-2, the Miscellaneous format includes 22 instructions with 12 slightly different formats; we use the extended opcode field, but a few of these instructions have immediate or register fields.

### *RV64G Core Instructions*

Almost every instruction found in the RV64G is found in the other architectures, as Figures K.9 through K.19 show. (For reference, definitions of the RISC-V instructions are found in Section A.9.) Instructions are listed under four categories: data transfer (Figure K.9); arithmetic, logical (Figure K.10); control (Figure K.11 and Figure K.12); and floating point (Figure K.13).

If a RV64G core instruction requires a short sequence of instructions in other architectures, these instructions are separated by semicolons in Figure K.9 through Figure K.13. (To avoid confusion, the destination register will always be the left-most operand in this appendix, independent of the notation normally used with each architecture.).

### *Compare and Conditional Branch*

Every architecture must have a scheme for compare and conditional branch, but despite all the similarities, each of these architectures has found a different way to perform the operation! Figure K.11 summarizes the control instructions, while Figure K.12 shows details of how conditional branches are handled. SPARC uses the traditional four condition code bits stored in the program status word: *negative*, *zero*, *carry*, and *overflow*. They can be set on any arithmetic or logical instruction; unlike earlier architectures, this setting is optional on each instruction. An explicit option leads to fewer problems in pipelined implementation. Although condition codes can be set as a side effect of an operation, explicit compares are synthesized with a subtract using  $r0$  as the destination. SPARC conditional branches test condition codes to determine all possible unsigned and signed relations. Floating point uses separate condition codes to encode the IEEE 754 conditions, requiring a floating-point compare instruction. Version 9 expanded SPARC branches in four ways: a separate set of condition codes for 64-bit operations; a branch that tests the contents of a register and branches if the value is  $=$ ,  $\neq$ ,  $<$ ,  $\leq$ ,  $>$ ,  $\geq$ , or  $\leq 0$ ; three more sets of floating-point condition codes; and branch instructions that encode static branch prediction.

Power also uses four condition codes: *less than*, *greater than*, *equal*, and *summary overflow*, but it has eight copies of them. This redundancy allows the Power instructions to use different condition codes without conflict, essentially giving Power eight extra 4-bit registers. Any of these eight condition codes can be the target of a compare instruction, and any can be the source of a conditional branch. The integer instructions have an option bit that behaves as if the integer is followed by a compare to zero that sets the first condition “register.” Power also lets the second “register” be optionally set by floating-point instructions. PowerPC provides logical operations among these eight 4-bit condition code registers (CRAND, CROR, CRXOR, CRNAND, CRNOR, CREQV), allowing more complex conditions to be tested by a single branch. Finally, Power includes a set of branch count registers, that are automatically decremented when tested, and can be used in a branch condition. There are also special instructions for moving from/to the condition register.

| Data transfer<br>(instruction formats) | R-I                                   | R-I                             | R-I, R-R         | R-I                                                                                                    | R-I, R-R              |
|----------------------------------------|---------------------------------------|---------------------------------|------------------|--------------------------------------------------------------------------------------------------------|-----------------------|
| Instruction name                       | ARMv8                                 | MIPS64                          | Power            | RV64G                                                                                                  | SPARC                 |
| Load byte signed/unsigned.             | LDR_B                                 | LB_                             | LBZ;<br>EXTSB    | LB_                                                                                                    | LD_B                  |
| Load halfword signed, unsigned         | LDR_H                                 | LH_                             | LHA/LHZ          | LH_                                                                                                    | LD_H                  |
| Load word                              | LDRSW/LDR                             | LW_                             | LW_              | LW_                                                                                                    | LD_W                  |
| Load double                            | LDRX                                  | LD                              | LD               | LD                                                                                                     | LD                    |
| Load float register SP/DP              | LD_                                   | L_C1                            | LF_              | FL_                                                                                                    | LD_F                  |
| Store byte                             | STB                                   | SB                              | STB              | SB                                                                                                     | STB                   |
| Store half word                        | STW                                   | SH                              | STH              | STH                                                                                                    | STH                   |
| Store word                             | STL                                   | SW                              | STW              | SW                                                                                                     | ST                    |
| Store double word                      | STX                                   | SD                              | SD               | SD                                                                                                     | STD                   |
| Store float SP/DP                      | ST_                                   | S_C1                            | STF_             | FS_                                                                                                    | ST_F                  |
| Load reserved                          | LDEXB,LDEXH<br>LDEXW,LDEXD            | LL, LLD                         | lwarx, ldarx, LR |                                                                                                        |                       |
| Store conditional                      | STEXB, STEXH, SC, SCD<br>STEXW, STEXD | stwcx, stdcx                    | SC               |                                                                                                        |                       |
| Read/write spec. register              | MF_, MT_                              | MF, MT_                         | M_SPR,           | csrr_,<br>csrr_i,                                                                                      | RD__,WR__             |
| Move integer to FP register            | ITOFS                                 | MFC1/<br>DMFC1                  | STW; LDFS        | STW; FLDWX                                                                                             | ST; LDF               |
| Move FP to integer register            | FTTOIS                                | MTC1/<br>DMTC1                  | STFS; LW         | FSTWX; LDW                                                                                             | STF; LD               |
| Synchronize data, instruction stream   | DSB<br>ISB                            | SYNC,<br>SYNCI                  | SYNC,<br>ISYNC   | Fence<br>Fence.i                                                                                       | MEMBAR<br>FLUSH       |
| Atomic operations                      | LDWAT, LDDAT<br>STWAT, STDAT          | LLWP,<br>LLDP,<br>SCWP,<br>SCDP |                  | AMOSWAP.W/D,<br>AMOADD.W.D,<br>AMOAND.W/D,<br>AMOXOR.W/D,<br>AMOOR.W/D,<br>AMOMIN_.W/D,<br>AMOMAX_.W/D | CASA, SWAP,<br>LDSTUB |

**Figure K.9 Desktop RISC data transfer instructions equivalent to RV64G core.** A sequence of instructions to synthesize a RV64G instruction is shown separated by semicolons. The MIPS and Power instructions for atomic operations load and conditionally store a pair of registers and can be used to implement the RV64G atomic operations with at most one intervening ALU instruction. The SPARC instructions: compare-and-swap, swap, LDSTUB provide atomic updates to a memory location and can be used to build the RV64G instructions. The Power3 instructions provide all the functionality, as the RV64G instructions, depending on a function field.

| <b>Arithmetic/logical<br/>(instruction formats)</b>                           | <b>R-R, R-I</b> | <b>R-R, R-I</b>        | <b>R-R, R-I</b> | <b>R-R, R-I</b>        | <b>R-R, R-I</b>  |
|-------------------------------------------------------------------------------|-----------------|------------------------|-----------------|------------------------|------------------|
| <b>Instruction name</b>                                                       | <b>ARM v8</b>   | <b>MIPS64</b>          | <b>Power v3</b> | <b>RISC-V</b>          | <b>SPARC v.9</b> |
| Add word, immediate                                                           | ADD, ADDI       | ADDU, ADDUI,           | ADD, ADDI       | ADDW, ADDWI            | ADD              |
| Add double word                                                               | ADDX            | DADDU, DADDUI          | ADD, ADDI       | ADD, ADDI              | ADD              |
| Subtract                                                                      | SUB, SUBI       | SUBU, SUBI             | SUBF            | SUBW, SUBWI            | SUB              |
| Subtract double word                                                          | SUBX            | DSUBU, DSUBUI          | SUBF            | SUB, SUBI              | SUB              |
| Multiply                                                                      | MUL, SMUL       | MUL, MULU, DMUL, DMULU | MULLW, MULLI    | MUL, MULU, MULW, MULWU | MULX             |
| Divide                                                                        | MULX, SMULX     | DIV, DIVU, DDIV, DDIVU | DIVW            | DIV, DIVU, DIVW, DIVWU | DIVX             |
| Remainder                                                                     |                 | MOD, MODU, DMOD, DMODU | MODSW, MODUW    | REM, REMU, REMW, REMWU |                  |
| And                                                                           | AND, ANDI       | AND, ANDI              | AND, ANDI       | AND, ANDI              | AND              |
| Or                                                                            | OR, ORI         | OR, ORI                | OR, ORI         | OR, ORI                | OR               |
| Xor                                                                           | XOR, XORI       | XOR, XORI              | XOR, XORI       | XOR, XORI              | XOR              |
| Load bits 31..16                                                              | MOV             | LUI                    | ADDIS           | ADDIS                  | SETHI<br>(Bfmt.) |
| Load upper bits of PC                                                         | ADR             | ADDIUPC                | ADDP CIS        | AUIPC                  |                  |
| Shift left logical, double word and word versions, immediate and variable     | LSL             | SLLV, SLL              | RLWINM          | SLL, SLLI, SLLW, SLLWI | SLL              |
| Shift right logical, double word and word version, immediate and variables    | RSL             | SRLV, SRL              | RLWINM<br>32-i  | SRL, SRLI, SRLW, SRLWI | SRL              |
| Shift right arithmetic, double word and word versions, immediate and variable | RSA             | SRAV, SRA              | SRAW            | SRA, SRAI, SRAW, SRAWI | SRA              |
| Compare                                                                       | CMP             | SLT/U, SL<br>TI/U      | CMP(I)CLR       | SLT/U,<br>SLTI/U       | SUBcc<br>r0, ... |

**Figure K.10** Desktop RISC arithmetic/logical instructions equivalent to RISC-V integer ISA. MIPS also provides instructions that trap on arithmetic overflow, which are synthesized in other architectures with multiple instructions. Note that in the “Arithmetic/logical” category all machines but SPARC use separate instruction mnemonics to indicate an immediate operand; SPARC offers immediate versions of these instructions but uses a single mnemonic. (Of course, these are separate opcodes!)

**K-14** ■ Appendix K *Survey of Instruction Set Architectures*

| Instruction name                 | ARMv8                | MIPS64                                                   | PowerPC                              | RISC-V                                      | SPARC v.9                          |
|----------------------------------|----------------------|----------------------------------------------------------|--------------------------------------|---------------------------------------------|------------------------------------|
| Branch on integer compare        | B.cond,<br>CBZ, CBNZ | BEQ, BNE, BC<br>B_Z (<, >,<br><=, >=)<br>OR<br>S***; BEZ | BEQ, BNE,<br>BLT, BGE,<br>BLTU, BGEU | BR_Z, BPCC<br>(<, >,<br><=, >=, =,<br>not=) |                                    |
| Branch on floating-point compare | B.cond               | BC1T,<br>BC1F                                            | BC                                   | BEZ, BNZ                                    | FBPfcc (<, >,<br><=,<br>>=, =,...) |
| Jump, jump register              | B, BR                | J, JR                                                    | B, BCLR,<br>BCCTR                    | JAL, JALR<br>(with x0)                      | BA, JMPL<br>r0,...                 |
| Call, call register              | BL, BLR              | JAL,<br>JALR                                             | BL, BLA,<br>BCLRL,<br>BCCTRL         | JAL, JALR                                   | CALL, JMPL                         |
| Trap                             | SVC, HVC,<br>SMC     | BREAK                                                    | TW, TWI                              | ECALL                                       | Ticc, SIR                          |
| Return from interrupt            | ERET                 | JR; ERET                                                 | RFI                                  | EBREAK                                      | DONE, RETRY,<br>RETURN             |

**Figure K.11** Desktop RISC control instructions equivalent to RV64G.

|                                                    | ARMv8                | MIPS64                   | PowerPC         | RISC-V                          | SPARC v.9                  |
|----------------------------------------------------|----------------------|--------------------------|-----------------|---------------------------------|----------------------------|
| Number of condition code bits (integer and FP)     | 16 (8 + the inverse) | none                     | 8 × 4 both      | none                            | 2 × 4 integer,<br>4 × 2 FP |
| Basic compare instructions (integer and FP)        | 1 integer; 1 FP      | 1 integer, 1 FP          | 4 integer, 2 FP | 2 integer; 3 FP                 | 1 FP                       |
| Basic branch instructions (integer and FP)         | 1                    | 2 integer, 1 FP          | 1 both          | 4 integer (used for FP as well) | 3 integer, 1 FP            |
| Compare register with register/constant and branch | —                    | =, not=                  | —               | =, not =, >=, <                 | —                          |
| Compare register to zero and branch                | —                    | =, not=, <, <=,<br>>, >= | —               | =, not=, <, <=,<br>>, >=        | =, not=, <, <=,<br>>, >=   |

**Figure K.12** Summary of five desktop RISC approaches to conditional branches. Integer compare on SPARC is synthesized with an arithmetic instruction that sets the condition codes using r0 as the destination.

RISC-V and MIPS are most similar. RISC-V uses a compare and branch with a full set of arithmetic comparisons. MIPS also uses compare and branch, but the comparisons are limited to equality and tests against zero. This limited set of conditions simplifies the branch determination (since an ALU operation is not required to test the condition), at the cost of sometimes requiring the use of a set-on-less-than instruction (SLT, SLTI, SLTU, SLTIU), which compares two operands and then set the destination register to 1 if less and to 0 otherwise. Figure K.12 provides

| Floating point (instruction formats)                                                                                        | R-R              | R-R                  | R-R                | R-R                   | R-R             |
|-----------------------------------------------------------------------------------------------------------------------------|------------------|----------------------|--------------------|-----------------------|-----------------|
| Instruction name                                                                                                            | ARMv8            | MIPS64               | PowerPC            | RISC-V                | SPARC v.9       |
| Add single, double                                                                                                          | FADD             | ADD.*                | FADD*              | FADD.*                | FADD*           |
| Subtract single, double                                                                                                     | FSUB             | SUB.*                | FSUB*              | FSUB.*                | FSUB*           |
| Multiply single, double                                                                                                     | FMUL             | MUL.*                | FMUL*              | FMUL.*                | FMUL*           |
| Divide single, double                                                                                                       | FDIV             | DIV.*                | FDIV*              | FDIV.*                | FDIV*           |
| Square root single, double                                                                                                  | FSQRT            | SQRT.*               | FSQRT*             | FSQRT.*               | FSQRT*          |
| Multiply add; Negative multiply add: single, double                                                                         | FMADD,<br>FNMADD | MADD.*<br>NMADD.*    | FMADD*,<br>FNMADD* | FMADD.*<br>FNMADD.*   |                 |
| Multiply subtract single, double, Negative multiply subtract: single, double                                                | FMSUB,<br>FNMSUB | MSUB.* ,<br>NMSUB.*  | FMSUB*,<br>FNMSUB* | FMSUB.* ,<br>FNMSUB.* |                 |
| Copy sign or negative sign double or single to another FP register                                                          | FMOV,<br>FNEG    | FMOV.* ,<br>FNEG.*   | FMOV*,<br>FNEG*    | FSGNJ.* ,<br>FSGNIN.* | FMOV*,<br>FNEG* |
| Replace sign bit with XOR of sign bits single double                                                                        | FABS             | FABS.*               | FABS*              | FSGNJP.*              | FABS*           |
| Maximum or minimum single, double                                                                                           | FMAX,<br>FMIN    | MAX.* ,<br>MIN.*     |                    | FMAX.* ,<br>FMIN.*    |                 |
| Classify floating point value single double                                                                                 |                  | CLASS.*              |                    | FCLASS.*              |                 |
| Compare                                                                                                                     | FCMP             | CMP.*                | FCMP*              | FCMP.*                | FCMP*           |
| Convert between FP single or double and FP single or double, OR integer single or double, signed and unsigned with rounding | FCVT             | CVT, CEIL,<br>FLLOOR |                    | FCVT                  | F*T0*           |

**Figure K.13 Desktop RISC floating-point instructions equivalent to RV64G ISA with an empty entry meaning that the instruction is unavailable.** ARMv8 uses the same assembly mnemonic for single and double precision; the register designator indicates the precision. “\*\*” is used as an abbreviation for S or D. For floating point compares all conditions: equal, not equal, less than, and less-than or equal are provided. Moves operate in both directions from/to integer registers. Classify sets a register based on whether the floating point quantity is plus or minus infinity, denorm,  $+/-0$ , etc.). The sign-injection instructions take two operands, but are primarily used to form floating point move, negate, and absolute value, which are separate instructions in the other ISAs.

additional details on conditional branch. RISC-V floating point comparisons sets an integer register to 0 or 1, and then use conditional branches on that content. MIPS also uses separate floating-point compare, which sets a floating point register to 0 or 1, which is then tested by a floating-point conditional branch.

ARM is similar to SPARC, in that it provides four traditional condition codes that are optionally set. CMP subtracts one operand from the other and the difference sets the condition codes. Compare negative (CMN) adds one operand to the other, and the sum sets the condition codes. TST performs logical AND on the two operands to set all condition codes but overflow, while TEQ uses exclusive OR to set the first three condition codes. Like SPARC, the conditional version of the ARM branch instruction tests condition codes to determine all possible unsigned and signed relations. ARMv8 added both bit-test instructions and also compare and branch against zero. Floating point compares on ARM, set the integer condition codes, which are used by the B.cond instruction.

As Figure K.13 shows the floating point support is similar on all five architectures.

### *RV64GC Core 16-bit Instructions*

Figures K.14 through K.17 summarize the data transfer, ALU, and control instructions for our three embedded processors: microMIPS64, RV64GC, and Thumb-2. Since these architectures are all based on 32-bit or 64-bit versions of the full architecture, we focus our attention on the functionality implemented by the 16-bit instructions. Since floating point is optional, we do not include it. I

| Instruction name                                     | microMIPS64<br>rs1;rs2/dst; offset | RV64GC<br>rs1;rs2/dst; offset | Thumb-2<br>rs1;rs2/dst; offset |
|------------------------------------------------------|------------------------------------|-------------------------------|--------------------------------|
| Load word                                            | 8;8;4                              | 8;8;5                         | 8;8;5                          |
| Load double word                                     |                                    | 8;8;5                         |                                |
| Load word with stack pointer as base register        | 1;32;5                             | 1;32;6                        | 1;3;8                          |
| Load double word with stack pointer as base register |                                    | 1;32;6                        |                                |
| Store word                                           | 8;8;4                              | 8;8;5                         | 8;8;5                          |
| Store double word                                    |                                    | 8;8;5                         |                                |
| Store word with stack pointer as base register       | 1;32;5                             | 1;32;6                        | 1;3;8                          |
| Store double with stack pointer as base register     |                                    | 1;32;6                        |                                |

**Figure K.14 Embedded RISC data transfer instructions equivalent to RV64GC 16-bit ISA; a blank indicates that the instruction is not a 16-bit instruction.** Rather than show the instruction name, where appropriate, we show the number of registers that can be the base register for the address calculation, followed by the number of registers that can be the destination for a load or the source for a store, and finally, the size of the immediate used for address calculation. For example: 8; 8; 5 for a load means that there are 8 possible base registers, 8 possible destination registers for the load, and a 5-bit offset for the address calculation. For a store, 8; 8; 5, specifies that the source of the value to store comes from one of 8 registers. Remember that Thumb-2 also has 32-bit instructions (although not the full ARMv8 set) and that RV64GC and microMIPS64 have the full set of 32-bit instructions in RV64I or MIPS64.

| Instruction Name/Function                       | microMIPS64        | RV64GC                  | Thumb-2            |
|-------------------------------------------------|--------------------|-------------------------|--------------------|
| Load immediate                                  | 8;7                | 32;6                    | 8;8                |
| Load upper immediate                            |                    | 32;6                    |                    |
| add immediate                                   | 32;4               | 32;6                    | 8;8;3              |
| add immediate word (32 bits) & sign extend      |                    | 32;6                    |                    |
| add immediate to stack pointer                  | 1;9                | 1;6<br>(adds 16x imm.)  | 1;7                |
| add immediate to stack pointer store in reg.    | 1;8;6              | 1;8;6<br>(adds 4x imm.) |                    |
| shift left/right logical                        | 8;8;3 (shift amt.) | 8;6(shift amt.)         | 8;8;5 (shift amt.) |
| shift right arithmetic                          |                    | 8;6(shift amt.)         | 8;8;5 (shift amt.) |
| AND immediate                                   | 8;8;4              | 8;6                     | 8;8                |
| move                                            | 32;32              | 32;32                   | 16;16              |
| add                                             | 8;8;8              | 32;32                   | 8;8;8<br>16;16     |
| AND, OR, XOR                                    | 8;8                | 8;8                     | 8;8                |
| subtract                                        | 8;8;8              | 8;8                     | 8;8;8              |
| add word, subtract word (32 bits) & sign extend |                    | 8;8                     |                    |

**Figure K.15 ALU instructions provided in RV64GC and the equivalents, if any, in the 16-bit instructions of micro-MIPS64 or Thumb-2.** An entry shows the number of register sources/destinations, followed by the size of the immediate field, if it exists for that instruction. The add to stack pointer with scaled immediate instructions are used for adjusting the stack pointer and creating a pointer to a location on the stack. In Thumb, the add has two forms one with three operands from the 8-register subset (Lo) and one with two operands but any of 16-registers.

|                                                | microMIPS64               | RV64GC                    | Thumb-2             |
|------------------------------------------------|---------------------------|---------------------------|---------------------|
| Unconditional branch                           | 10-bit offset             | 11-bit offset             | 11-bit offset       |
| Unconditional branch and link                  |                           | 11-bit offset             | 11-bit offset       |
| Unconditional branch to register w/wo link     | any of 32 registers       | any of 32 registers       |                     |
| Compare register to zero ( $=/!=$ ) and branch | 8 registers; 7-bit offset | 8 registers; 8-bit offset | no: but see caption |

**Figure K.16 Summary of three embedded RISC approaches to conditional branches.** A blank indicates that the instruction does not exist. Thumb-2 uses 4 condition code bits; it provides a conditional branch that tests the 4-bit condition code and has a branch offset of 8 bits.

| Function                        | Definition                                         | ARMv8                    | MIPS64             | PowerPC                               | SPARC v.9 |
|---------------------------------|----------------------------------------------------|--------------------------|--------------------|---------------------------------------|-----------|
| Load/store multiple registers   | Loads or stores 2 or more registers                | Load pair,<br>store pair |                    | Load store multiple (<=31 registers), |           |
| Cache manipulation and prefetch | Modifies status of a cache line or does a prefetch | Prefetch                 | CACHE,<br>PREFETCH | Prefetch                              | Prefetch  |

**Figure K.17 Data transfer instructions not found in RISC-V core but found in two or more of the five desktop architectures.** SPARC requires memory accesses to be aligned, while the other architectures support unaligned access, albeit, often with major performance penalties. The other architectures do not require alignment, but may use slow mechanisms to handle unaligned accesses. MIPS provides a set of instructions to handle misaligned accesses: LDL and LDR (load double left and load double right instructions) work as a pair to load a misaligned word; the corresponding store instructions perform the inverse. The Prefetch instruction causes a cache prefetch, while CACHE provides limited user control over the cache state.

### Instructions: Common Extensions beyond RV64G

Figures K.15 through K.18 list instructions not found in Figures K.9 through K.13 in the same four categories (data transfer, ALU, and control). The only significant floating point extension is the reciprocal instruction, which both MIPS64 and Power support. Instructions are put in these lists if they appear in more than one of the standard architectures. Recall that Figure K.3 on page 6 showed the address modes supported by the various instruction sets. All three processors provide more address modes than provided by RV64G. The loads and stores using these additional address modes are not shown in Figure K.17, but are effectively additional data transfer instructions. This means that ARM has 64 additional load and store instructions, while Power3 has 12, and MIPS64 and SPARVv9 each have 4.

To accelerate branches, modern processors use dynamic branch prediction (see Section 3.3). Many of these architectures in earlier versions supported delayed branches, although they have been dropped or largely eliminated in later versions

| Name             | Definition                                 | ARMv8 | MIPS64                                     | PowerPC             | SPARC v.9            |
|------------------|--------------------------------------------|-------|--------------------------------------------|---------------------|----------------------|
| Delayed branches | Delayed branches with/without cancellation |       | BEQ, BNE, BGTZ,<br>BLEZ, BCxEQZ,<br>BCxNEZ |                     | BPcc, A,<br>FPBcc, A |
| Conditional trap | Traps if a condition is true               |       | TEQ, TNE, TGE,<br>TLT, TGEU, TLTU          | TW, TD,<br>TWI, TDI | Tcc                  |

**Figure K.18 Control instructions not found in RV64G core but found in two or more of the other architectures.** MIPS64 Release 6 has nondelayed and normal delayed branches, while SPARC v.9 has delayed branches with cancellation based on the static prediction.

of the architecture, typically by offering a nondelayed version, as the preferred conditional branch. The SPARC “annulling” branch is an optimized form of delayed branch that executes the instruction in the delay slot only if the branch is taken; otherwise, the instruction is annulled. This means the instruction at the target of the branch can safely be copied into the delay slot since it will only be executed if the branch is taken. The restrictions are that the target is not another branch and that the target is known at compile time. (SPARC also offers a nondelayed jump because an unconditional branch with the annul bit set does *not* execute the following instruction.).

In contrast to the differences among the full ISAs, the 16-bit subsets of the three embedded ISAs have essentially no significant differences other than those described in the earlier figures (e.g. size of immediate fields, uses of SP or other registers, etc.).

Now that we have covered the similarities, we will focus on the unique features of each architecture. We first cover the desktop/server RISCs, ordering them by length of description of the unique features from shortest to longest, and then the embedded RISCs.

### Instructions Unique to MIPS64 R6

MIPS has gone through six generations of instruction sets. Generations 1–4 mostly added instructions. Release 6 eliminated many older instructions but also provided support for nondelayed branches and misaligned data access. Figure K.19 summarizes the unique instructions in MIPS64 R6.

| Instruction class | Instruction name(s)                 | Function                                                                                                                                    |
|-------------------|-------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| ALU               | Byte align                          | Take a pair of registers and extract a word or double word of bytes.<br>Used to implement unaligned byte copies.                            |
|                   | Align Immediate to PC               | Adds the upper 16 bits of the PC to an immediate shifted left 16 bits and puts the result in a register; Used to get a PC-relative address. |
|                   | Bit swap                            | Reverses the bits in each byte of a register.                                                                                               |
|                   | No-op and link                      | Puts the value of PC+8 into a register                                                                                                      |
|                   | Logical NOR                         | Computes the NOR of 2 registers                                                                                                             |
| Control transfer  | Branch and Link conditional         | Compares a register to 0 and does a branch if condition is true; places the return address in the link register.                            |
|                   | Jump indexed, Jump and link indexed | Adds an offset and register to get new PC, w/wo link address                                                                                |

**Figure K.19** Additional instructions provided MIPS64 R6. In addition, there are several instructions for supporting virtual machines, most are privileged.

## Instructions Unique to SPARC v.9

Several features are unique to SPARC. We review the major figures and then summarize those and small differences in a figure.

### *Register Windows*

The primary unique feature of SPARC is register windows, an optimization for reducing register traffic on procedure calls. Several banks of registers are used, with a new one allocated on each procedure call. Although this could limit the depth of procedure calls, the limitation is avoided by operating the banks as a circular buffer. The knee of the cost-performance curve seems to be six to eight banks; programs with deeper call stacks, would need to save and restore the registers to memory.

SPARC can have between 2 and 32 windows, typically using 8 registers each for the globals, locals, incoming parameters, and outgoing parameters. (Given that each window has 16 unique registers, an implementation of SPARC can have as few as 40 physical registers and as many as 520, although most have 128 to 136, so far.) Rather than tie window changes with call and return instructions, SPARC has the separate instructions `SAVE` and `RESTORE`. `SAVE` is used to “save” the caller’s window by pointing to the next window of registers in addition to performing an add instruction. The trick is that the source registers are from the caller’s window of the addition operation, while the destination register is in the callee’s window. SPARC compilers typically use this instruction for changing the stack pointer to allocate local variables in a new stack frame. `RESTORE` is the inverse of `SAVE`, bringing back the caller’s window while acting as an add instruction, with the source registers from the callee’s window and the destination register in the caller’s window. This automatically deallocates the stack frame. Compilers can also make use of it for generating the callee’s final return value.

The danger of register windows is that the larger number of registers could slow down the clock rate. This was not the case for early implementations. The SPARC architecture (with register windows) and the MIPS R2000 architecture (without) have been built in several technologies since 1987. For several generations the SPARC clock rate has not been slower than the MIPS clock rate for implementations in similar technologies, probably because cache access times dominate register access times in these implementations. With the advent of multiple issue, which requires many more register ports, as well as register renaming or reorder buffers, register windows posed a larger penalty. Register windows were a feature of the original Berkeley RISC designs, and their inclusion in SPARC was inspired by those designs. Tensilica is the only other major architecture in use today employs them, and they were not included in the RISC-V ISA.

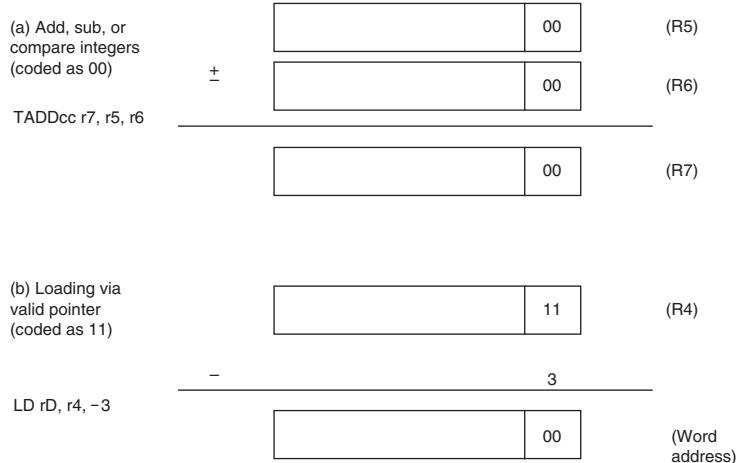
### *Fast Traps*

SPARCv9 includes support to make traps fast. It expands the single level of traps to at least four levels, allowing the window overflow and underflow trap handlers to be interrupted. The extra levels mean the handler does not need to check for page faults or

misaligned stack pointers explicitly in the code, thereby making the handler faster. Two new instructions were added to return from this multilevel handler: RETRY (which retries the interrupted instruction) and DONE (which does not). To support user-level traps, the instruction RETURN will return from the trap in nonprivileged mode.

### *Support for LISP and Smalltalk*

The primary remaining arithmetic feature is tagged addition and subtraction. The designers of SPARC spent some time thinking about languages like LISP and Smalltalk, and this influenced some of the features of SPARC already discussed: register windows, conditional trap instructions, calls with 32-bit instruction addresses, and multi-word arithmetic (see Taylor et al. [1986] and Ungar et al. [1984]). A small amount of support is offered for tagged data types with operations for addition, subtraction, and hence comparison. The two least-significant bits indicate whether the operand is an integer (coded as 00), so TADDcc and TSUBcc set the overflow bit if either operand is not tagged as an integer or if the result is too large. A subsequent conditional branch or trap instruction can decide what to do. (If the operands are not integers, software recovers the operands, checks the types of the operands, and invokes the correct operation based on those types.) It turns out that the misaligned memory access trap can also be put to use for tagged data, since loading from a pointer with the wrong tag can be an invalid access. Figure K.20 shows both types of tag support.



**Figure K.20** SPARC uses the two least-significant bits to encode different data types for the tagged arithmetic instructions. (a) Integer arithmetic, which takes a single cycle as long as the operands and the result are integers. (b) The misaligned trap can be used to catch invalid memory accesses, such as trying to use an integer as a pointer. For languages with paired data like LISP, an offset of  $-3$  can be used to access the even word of a pair (CAR) and  $+1$  can be used for the odd word of a pair (CDR).

| Instruction class           | Instruction name(s)                                | Function                                                                                                      |
|-----------------------------|----------------------------------------------------|---------------------------------------------------------------------------------------------------------------|
| Data transfer               | SAVE, RESTORE                                      | Save or restore a register window                                                                             |
|                             | Nonfaulting load                                   | Version of load instructions that do not generate faults on address exceptions; allows speculation for loads. |
| ALU                         | Tagged add, Tagged subtract, with and without trap | Perform a tagged add/subtract, set condition codes, optionally trap.                                          |
| Control transfer            | Retry, Return, and Done                            | To provide handling for traps.                                                                                |
| Floating Point Instructions | FMOVcc                                             | Conditional move between FP registers based on integer or FP condition codes.                                 |

**Figure K.21 Additional instructions provided in SPARCv9.** Although register windows are by far the most significant distinction, they do not require many instructions!

Figure K.21 summarizes the additional instructions mentioned above as well as several others.

### Instructions Unique to ARM

Earlier versions of the ARM architecture (ARM v6 and v7) had a number of unusual features including conditional execution of all instructions, and making the PC a general purpose register. These features were eliminated with the arrival of ARMv8 (in both the 32-bit and 64-bit ISA). What remains, however, is much of the complexity, at least in terms of the size of the instruction set. As Figure K.3 on page 6 shows, ARM has the most addressing modes, including all those listed in the table; remember that these addressing modes add dozens of load/store instructions compared to RVG, even though they are not listed in the table that follows. As Figure K.6 on page 8 shows, ARMv8 also has by far the largest number of different instruction formats, which reflects a variety of instructions, as well as the different addressing modes, some of which are applicable to some loads and stores but not others.

Most ARMv8 ALU instructions allow the second operand to be shifted before the operation is completed. This extends the range of immediates, but operand shifting is not limited to immediates. The shift options are shift left logical, shift right logical, shift right arithmetic, and rotate right. In addition, as in Power3, most ALU instructions can optionally set the condition flags. Figure K.22 includes the additional instructions, but does not enumerate all the varieties (such as optional setting of the condition flags); see the caption for more detail. While conditional execution of all instructions was eliminated, ARMv8 provides a number of conditional instructions beyond the conditional move and conditional set, mentioned earlier.

| Instruction class | Instruction name(s)                                                    | Function                                                                                                              |
|-------------------|------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------|
| Data transfer     | Load/Store Non-temporal pair                                           | Loads/stores a pair of registers with an indication not to cache the data. Base + scaled offset addressing mode only. |
| ALU               | Add Extended word/double word                                          | Add 2 registers after left shifting the second register operand and extending it.                                     |
|                   | Add with shift; add immediate with shift                               | Adds with shift of the second operand.                                                                                |
|                   | Address of page                                                        | Computes the address of a page based on PC (similar to ADDUIPC, which is the same as ADR in ARMv8)                    |
|                   | AND, OR, XOR, XOR NOT shifted register                                 | Logical operation on a register and a shifted register.                                                               |
|                   | Bit field clear shifted                                                | Shift operand, invert and AND with another operand                                                                    |
|                   | Conditional compare, immediate, negative, negative immediate           | If condition true, then set condition flags to compare result, otherwise leave condition flags untouched.             |
|                   | Conditional increment, invert, negate                                  | If condition then set destination to increment/invert/negate of source register                                       |
|                   | CRC                                                                    | Computes a CRC checksum: byte, word, halfword, double                                                                 |
|                   | Multiply add, subtract                                                 | Integer multiply-add or multiply-subtract                                                                             |
|                   | Multiply negate                                                        | Negate the product of two integers; word & double word                                                                |
|                   | Move immediate or inverse                                              | Replace 16-bits in a register with immediate, possibly shifted                                                        |
|                   | Reverse bit order                                                      | Reverses the order of bits in a register                                                                              |
|                   | Signed bit field move                                                  | Move a signed bit field; sign extend to left; zero extend to right                                                    |
|                   | Unsigned divide, multiple, multiply negate, multiply-add, multiply-sub | Unsigned versions of the basic instructions                                                                           |
| Control transfer  | CBNZ, CBZ                                                              | Compare branch $=!= 0$ , indicating this is not a call or return.                                                     |
|                   | TBNZ, TBZ                                                              | Tests bit in a register $=!= 0$ , and branch.                                                                         |

**Figure K.22 Additional instructions provided in ARMv8, the AArch64 instruction set.** Unless noted the instruction is available in a word and double word format, if there is a difference. Most of the ALU instructions can optionally set the condition codes; these are not included as separate instructions here or in earlier tables.

### Instructions Unique to Power3

Power3 is the result of several generations of IBM commercial RISC machines—IBM RT/PC, IBM Power1, and IBM Power2, and the PowerPC development, undertaken primarily by IBM and Motorola. First, we describe branch registers and the support for loop branches. Figure K.23 then lists the other instructions provided only in Power3.

| Instruction class           | Instruction name(s)                                          | Function                                                                                                                                                                     |
|-----------------------------|--------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Data transfer               | LHBRX, LWBRX, LDBRX                                          | Loads a halfword/word/double word but reverses the byte order.                                                                                                               |
|                             | SHBRX, SWBRX, SDBRX                                          | Stores a halfword/word/double word but reverses the byte order                                                                                                               |
|                             | LDQ, STQ                                                     | Load/store quadword to a register pair.                                                                                                                                      |
| ALU                         | DRAN                                                         | Generate a random number in a register                                                                                                                                       |
|                             | CMPB                                                         | Compares the individual bytes in a register and sets another register byte by byte.                                                                                          |
|                             | CMPRB                                                        | Compares a byte (x) against two other bytes (y and z) and sets a condition to indicate if the value of $y \leq x \leq z$ .                                                   |
|                             | CRAND, CRNAND, CROR,<br>CRNOR, CRXOR, CREQV,<br>CORC, CRANDC | Logical operations on the condition register.                                                                                                                                |
|                             | ZCMPEQB                                                      | Compares a byte (x) against the eight bytes in another register and sets a condition to indicate if $x = \text{any of the 8 bytes}$                                          |
|                             | EXTSWSL                                                      | Sign extend word and shift left                                                                                                                                              |
|                             | POPCNTB, POPCNTW<br>POPCNTD                                  | Count number of 1s in each byte and place total in another byte.<br>Count number of 1s in each word and place total in another word.<br>Count number of 1s in a double word. |
|                             | PRTYD, PRTYW                                                 | Compute byte parity of the bytes in a word or double word.                                                                                                                   |
|                             | BPERMD                                                       | Permutates the bits in a double word, producing a permuted byte.                                                                                                             |
|                             | CDTBCD, CDCBCD,<br>ADDGCS                                    | Instructions to convert from/to binary coded decimal (BCD) or operate on two BCD values                                                                                      |
| Control transfer            | BA, BCA                                                      | Branches to an absolute address, conditionally & unconditionally                                                                                                             |
|                             | BCCTR, BCCTRL                                                | Conditional branch to address in the count register, w/wo linking                                                                                                            |
|                             | BCTSAR, BCTARL                                               | Conditional branch to address in the Branch Target Address register, w/wo linking                                                                                            |
|                             | CLRBHRB, MFBHRBE                                             | Manipulate the branch history rolling buffer.                                                                                                                                |
| Floating Point Instructions | FRSQRT                                                       | Computes an estimate of reciprocal of the square root,                                                                                                                       |
|                             | FTDIV, FTSQRT                                                | Tests for divide by zero or square of negative number                                                                                                                        |
|                             | FSEL                                                         | Test register against zero and select one of two operands to move                                                                                                            |
|                             | Decimal floating point operations                            | A series of 48 instructions to support decimal floating point.                                                                                                               |

**Figure K.23 Additional instructions provided in Power3.** Rotate instructions have two forms: one that sets a condition register and one that does not. There are a set of string instructions that load up to 32 bytes from an arbitrary address to a set of registers. These instructions will be phased out in future implementations, and hence we just mention them here.

### *Branch Registers: Link and Counter*

Rather than dedicate one of the 32 general-purpose registers to save the return address on procedure call, Power3 puts the address into a special register called the *link register*. Since many procedures will return without calling another procedure, link doesn't always have to be saved away. Making the return address a special register makes the return jump faster since the hardware need not go through the register read pipeline stage for return jumps.

In a similar vein, Power3 has a *count register* to be used in for loops where the program iterates for a fixed number of times. By using a special register the branch hardware can determine quickly whether a branch based on the count register is likely to branch, since the value of the register is known early in the execution cycle. Tests of the value of the count register in a branch instruction will automatically decrement the count register.

Given that the count register and link register are already located with the hardware that controls branches, and that one of the problems in branch prediction is getting the target address early in the pipeline (see Appendix C), the Power architects decided to make a second use of these registers. Either register can hold a target address of a conditional branch. Thus, PowerPC supplements its basic conditional branch with two instructions that get the target address from these registers (BCLR, BCCTR). Figure K.23 shows the several dozen instructions that have been added; note that there is an extensive facility for decimal floating point, as well.

## **Instructions: Multimedia Extensions of the Desktop/Server RISCs**

Support for multimedia and graphics operations developed in several phases, beginning in 1996 with Intel MMX, MIPS MDMX, and SPARC VIS. As described in Section 4.3, which we assume the reader has read, these extensions allowed a register to be treated as multiple independent small integers (8 or 16 bits long) with arithmetic and logical operations done in parallel on all the items in a register. These initial SIMD extensions, sometimes called packed SIMD, were further developed after 2000 by widening the registers, partially or totally separating them from the general purpose or floating pointer registers, and by adding support for parallel floating point operations. RISC-V has reserved an extension for such packed SIMD instructions, but the designers have opted to focus on a true vector extension for the present. The vector extension RV64V is a vector architecture, and, as Section 4.3 points out, a true vector instruction set is considerably more general, and can typically perform the operations handled by the SIMD extensions using vector operations.

Figure K.24 shows the basic structure of the SIMD extensions in ARM, MIPS, Power, and SPARC. Note the difference in how the SIMD “vector registers” are structured: repurposing the floating point, extending the floating point, or adding additional registers. Other key differences include support for FP as well as integers,

|                                               | <b>ARMv8</b>                       | <b>MIPS64 R6</b>                    | <b>Power v3.0</b>  | <b>SPARCv9</b>       |
|-----------------------------------------------|------------------------------------|-------------------------------------|--------------------|----------------------|
| Name of ISA extension                         | Advanced SIMD                      | MIPS64 SIMD Architecture            | Vector Facility    | VIS                  |
| Date of Current Version                       | 2011                               | 2012                                | 2015               | 1995                 |
| Vector registers: # x size                    | 32 x 128 bits                      | 32 x 128 bits                       | 32 x 128 bits      | 32 x 64 bits         |
| Use GP/FP registers or independent set        | extend FP registers doubling width | extend FP registers doubling width  | Independent        | Same as FP registers |
| Integer data sizes                            | 8, 16, 32, 64                      | 8, 16, 32, 64                       | 8, 16, 32, 64, 128 | 8, 16, 32            |
| FP data sizes                                 | 32, 64                             | 32, 64                              | 32                 |                      |
| Immediates for integer and logical operations |                                    | 5 bits arithmetic<br>8 bits logical |                    |                      |

**Figure K.24 Structure of the SIMD extensions intended for multimedia support.** In addition to the vector facility, The last row states whether the SIMD instruction set supports immediates (e.g, add vector immediate or AND vector immediate); the entry states the size of immediates for those ISAs that support them. Note that the fact that an immediate is present is encoded in the opcode space, and could alternatively be added to the next table as additional instructions. Power 3 has an optional Vector-Scalar Extension. The Vector-Scalar Extension defines a set of vector registers that overlap the FP and normal vector registers, eliminating the need to move data back and forth to the vector registers. It also supports double precision floating point operations.

support for 128-bit integers, and provisions for immediate fields as operands in integer and logical operations. Standard load and store instructions are used for moving data from the SIMD registers to memory with special extensions to handle moving less than a full SIMD register. SPARC VIS, which was one of the earliest ISA extensions for graphics, is much more limited: only add, subtract, and multiply are included, there is no FP support, and only limited instructions for bit element operations; we include it in Figure K.24 but will not be going into more detail.

Figure K.25 shows the arithmetic instructions included in these SIMD extensions; only those appearing in at least two extensions are included. MIPS SIMD includes many other instructions, as does the Power 3 Vector-Scalar extension, which we do not cover. One frequent feature not generally found in general-purpose microprocessors is saturating operations. Saturation means that when a calculation overflows the result is set to the largest positive number or most negative number, rather than a modulo calculation as in two's complement arithmetic. Commonly found in digital signal processors (see the next subsection), these saturating operations are helpful in routines for filtering. Another common extension are instructions for accumulating values within a single register; the dot product instruction and the maximum/minimum instructions are typical examples.

In addition to the arithmetic instructions, the most common additions are logical and bitwise operations and instructions for doing version of permutations and packing elements into the SIMD registers. These additions are summarized in Figure K.26. Lastly, all three extensions support SIMD FP operations, as summarized in Figure K.27.

| Instruction category                  | ARM Advanced SIMD | MIPS SIMD        | Power Vector Facility |
|---------------------------------------|-------------------|------------------|-----------------------|
| Add/subtract                          | 16B, 8H, 4W; 2 D  | 16B, 8H, 4W; 2 D | 16B, 8H, 4W, 2 D, Q   |
| Saturating add/sub                    | 16B, 8H, 4W; 2 D  | 16B, 8H, 4W; 2 D | 16B, 8H, 4W, 2 D, Q   |
| Absolute value of difference          | 16B, 8H, 4W; 2 D  | 16B, 8H, 4W; 2 D | 16B, 8H, 4W; 2 D; Q   |
| Adjacent add & subtract (pairwise)    | 16B, 8H, 4W       | 16B, 8H, 4W      | 16B, 8H, 4W; 2 D      |
| Average                               |                   | 16B, 8H, 4W; 2 D | 16B, 8H, 4W; 2 D; Q   |
| Dot product add, dot product subtract | 16B, 8H, 4W       | 16B, 8H, 4W      | 16B, 8H, 4W; 2 D      |
| Divide: signed, unsigned              | 16B, 8H, 4W       | 16B, 8H, 4W; 2 D | 16B, 8H, 4W; 2 D; Q   |
| Multiply: signed, unsigned            | 16B, 8H, 4W       | 16B, 8H, 4W      | 16B, 8H, 4W; 2 D      |
| Multiply add, multiply subtract       | 16B, 8H, 4W       | 16B, 8H, 4W      | 16B, 8H, 4W; 2 D      |
| Maximum, signed & unsigned            | 16B, 8H, 4W; 2 D  | 16B, 8H, 4W; 2 D | 16B, 8H, 4W; 2 D; Q   |
| Minimum, signed & unsigned            | 16B, 8H, 4W; 2 D  | 16B, 8H, 4W; 2 D | 16B, 8H, 4W; 2 D; Q   |
| Modulo, signed & unsigned             |                   | 16B, 8H, 4W; 2 D | 16B, 8H, 4W; 2 D; Q   |
| Compare equal                         | 16B, 8H, 4W; 2 D  | 16B, 8H, 4W; 2 D | 16B, 8H, 4W; 2 D; Q   |
| Compare <, <=, signed, unsigned       | 16B, 8H, 4W; 2 D  | 16B, 8H, 4W; 2 D | 16B, 8H, 4W; 2 D; Q   |

**Figure K.25 Summary of arithmetic SIMD instructions.** B stands for byte (8 bits), H for half word (16 bits), and W for word (32 bits), D for double word (64 bits), and Q for quad word (128 bits). Thus, 8B means an operation on 8 bytes in a single instruction. Note that some instructions—such as adjacent add/subtract, or multiply—produce results that are twice the width of the inputs (e.g. multiply on 16 bytes produces 8 halfword results). Dot product is a multiply and accumulate. The SPARC VIS instructions are aimed primarily at graphics and are structured accordingly.

| Instruction category                  | ARM Advanced SIMD | MIPS SIMD           | Power Vector Facility |
|---------------------------------------|-------------------|---------------------|-----------------------|
| Shift right/left, logical, arithmetic | 16B, 8H, 4W; 2 D  | 16B, 8H, 4W; 2 D; Q | 16B, 8H, 4W; 2 D; Q   |
| Count leading or trailing zeros       | 16B, 8H, 4W; 2 D  | 16B, 8H, 4W; 2 D    | 16B, 8H, 4W; 2 D; Q   |
| and/or/xor                            | Q                 | Q                   | Q                     |
| Bit insert & extract                  | 16B, 8H, 4W; 2 D  | 16B, 8H, 4W; 2 D    | 16B, 8H, 4W; 2 D; Q   |
| Population count                      |                   | 16B, 8H, 4W; 2 D    | 16B, 8H, 4W; 2 D; Q   |
| Interleave even/odd, left/right       |                   | 16B, 8H, 4W; 2 D    | 6B, 8H, 4W; 2 D       |
| Pack even/odd                         |                   | 16B, 8H, 4W; 2 D    | 6B, 8H, 4W; 2 D       |
| Shuffle                               |                   | 16B, 8H, 4W; 2 D    | 16B, 8H, 4W; 2 D      |
| SPLAT                                 |                   | 16B, 8H, 4W; 2 D    | 16B, 8H, 4W; 2 D      |

**Figure K.26 Summary of logical, bitwise, permute, and pack/unpack instructions, using the same format as the previous figure.** When there is a single operand the instruction applies to the entire register; for logical operations there is no difference. Interleave puts together the elements (all even, odd, leftmost or rightmost) from two different registers to create one value; it can be used for unpacking. Pack moves the even or odd elements from two different registers to the leftmost and rightmost halves of the result. Shuffle creates a from two registers based on a mask that selects which source for each item. SPLAT copies a value into each item in a register.

| Instruction category               | ARM Advanced SIMD | MIPS SIMD | Power Vector Facility |
|------------------------------------|-------------------|-----------|-----------------------|
| FP add, subtract, multiply, divide | 4W, 2D            | 4W, 2D    | 4W, 2D                |
| FP multiply add/subtract           | 4W, 2D            | 4W, 2D    | 4W, 2D                |
| FP maximum/minimum                 | 4W, 2D            | 4W, 2D    | 4W, 2D                |
| FP SQRT and 1/SQRT                 | 4W, 2D            | 4W, 2D    | 4W, 2D                |
| FP Compare                         | 4W, 2D            | 4W, 2D    | 4W, 2D                |
| FP Convert to/from integer         | 4W, 2D            | 4W, 2D    | 4W, 2D                |

**Figure K.27** Summary of floating point, using the same format as the previous figure.

### Instructions: Digital Signal-Processing Extensions of the Embedded RISCs

Both Thumb2 and microMIPS32 provide instructions for DSP (Digital Signal Processing) and multimedia operations. In Thumb2, these are part of the core instruction set; in microMIPS32, they are part of the DSP extension. These extensions, which are encoded as 32-bit instructions, are less extensive than the multimedia and graphics support provided in the SIMD/Vector extensions of MIPS64 or ARMv8 (AArch64). Like those more comprehensive extensions, the ones in Thumb2 and microMIPS32 also rely on packed SIMD, but they use the existing integer registers, with a small extension to allow a wide accumulator, and only operate on integer data. RISC-V has specified that the “P” extension will support packed integer SIMD using the floating point registers, but at the time of publication, the specification was not completed.

DSP operations often include linear algebra functions and operations such as convolutions; these operations produce intermediate results that will be larger than the inputs. In Thumb2, this is handled by a set of operations that produce 64-bit results using a pair of integer registers. In microMIPS32 DSP, there are 4 64-bit accumulator registers, including the Hi-Lo register, which already exists for doing integer multiply and divide. Both architectures provide parallel arithmetic using bytes, halfwords, and words, as in the multimedia extensions in ARMv8 and MIPS64. In addition, the MIPS DSP extension handles fractional data, such data is heavily used in DSP operations. Fractional data items have a sign bit and the remaining bits are used to represent the fraction, providing a range of values from -1.0 to 0.9999 (in decimal). MIPS DSP supports two fractional data sizes Q15 and Q31 each with one sign bit and 15 or 31 bits of fraction.

Figure K.28 shows the common operations using the same notation as was used in Figure K.25. Remember that the basic 32-bit instruction set provides additional functionality, including basic arithmetic, logical, and bit manipulation.

| Function                                                                                                                       | Thumb-2 | microMIPS32 DSP    |
|--------------------------------------------------------------------------------------------------------------------------------|---------|--------------------|
| Add/Subtract                                                                                                                   | 4B, 2H  | 4B, 2Q15           |
| Add /Subtract with saturation                                                                                                  | 4B, 2H  | 4B, 2Q15, Q31      |
| Add/Subtract with Exchange (exchanges halfwords in rt, then adds first halfword and subtracts second) with optional saturation | 2H      |                    |
| Reduce by add (sum the values)                                                                                                 | 4B      |                    |
| Absolute value                                                                                                                 |         | 2Q15, Q31          |
| Precision reduce/increase (reduces or increases the precision of a value)                                                      |         | 2B, Q15, 2Q15, Q31 |
| Shifts: left, right, logical & arithmetic, with optional saturation                                                            | 4B, 2H  |                    |
| Multiply                                                                                                                       | 2H      | 2B, 2H, 2Q15       |
| Multiply add/subtract (to GPR or accumulator register in MIPS)                                                                 | 2H      | 2Q15               |
| Complex multiplication step (2 multiplies and addition/subtraction)                                                            | 2H      | 2Q15               |
| Multiply and accumulate (by addition or subtraction)                                                                           | 2H      | Q15, Q31           |
| Replicate bits                                                                                                                 |         | B, H               |
| Compare: =, <, <=, sets condition field                                                                                        | 4B, 2H  |                    |
| Pick (use condition bits to choose bytes or halfwords from two operands)                                                       | 4B, 2H  |                    |
| Pack choosing a halfword from each operand                                                                                     |         | H                  |
| Extract                                                                                                                        |         | Q63                |
| Move from/to accumulator                                                                                                       |         | DW                 |

**Figure K.28 Summary of two embedded RISC DSP operations, showing the data types for each operation.** A blank indicates that the operation is not supported as a single instruction. Byte quantities are usually unsigned. Complex multiplication step implements multiplication of complex numbers where each component is a Q15 value. ARM uses its standard condition register, while MIPS adds a set of condition bits as part of the state in the DSP extension.

## Concluding Remarks

This survey covers the addressing modes, instruction formats, and almost all the instructions found in 8 RISC architectures. Although the later sections concentrate on the differences, it would not be possible to cover 8 architectures in these few pages if there were not so many similarities. In fact, we would guess that more than 90% of the instructions executed for any of these architectures would be found in Figures K.9 through K.13. To contrast this homogeneity, Figure K.29 gives a summary for four architectures from the 1970s in a format similar to that shown in Figure K.1. (Since it would be impossible to write a single section in this style for those architectures, the next three sections cover the 80x86, VAX, and IBM 360/370.) In the history of computing, there has never been such widespread agreement on computer architecture as there has been since the RISC ideas emerged in the 1980s.

|                                         | <b>IBM 360/370</b>              | <b>Intel 8086</b>                    | <b>Motorola 68000</b>                    | <b>DEC VAX</b>          |
|-----------------------------------------|---------------------------------|--------------------------------------|------------------------------------------|-------------------------|
| Date announced                          | 1964/1970                       | 1978                                 | 1980                                     | 1977                    |
| Instruction size(s) (bits)              | 16, 32, 48                      | 8, 16, 24, 32, 40, 48                | 16, 32, 48, 64, 80                       | 8, 16, 24, 32, ..., 432 |
| Addressing (size, model)                | 24 bits, flat/<br>31 bits, flat | 4 + 16 bits,<br>segmented            | 24 bits, flat                            | 32 bits, flat           |
| Data aligned?                           | Yes 360/No 370                  | No                                   | 16-bit aligned                           | No                      |
| Data addressing modes                   | 2/3                             | 5                                    | 9                                        | =14                     |
| Protection                              | Page                            | None                                 | Optional                                 | Page                    |
| Page size                               | 2 KB & 4 KB                     | —                                    | 0.25 to 32 KB                            | 0.5 KB                  |
| I/O                                     | Opcode                          | Opcode                               | Memory mapped                            | Memory mapped           |
| Integer registers (size, model, number) | 16 GPR $\times$ 32 bits         | 8 dedicated<br>data $\times$ 16 bits | 8 data and 8 address<br>$\times$ 32 bits | 15 GPR $\times$ 32 bits |
| Separate floating-point registers       | 4 $\times$ 64 bits              | Optional: 8 $\times$ 80 bits         | Optional: 8 $\times$ 80 bits             | 0                       |
| Floating-point format                   | IBM (floating<br>hexadecimal)   | IEEE 754 single,<br>double, extended | IEEE 754 single,<br>double, extended     | DEC                     |

**Figure K.29 Summary of four 1970s architectures.** Unlike the architectures in Figure K.1, there is little agreement between these architectures in any category. (See Section K.3 for more details on the 80x86 and Section K.4 for a description of the VAX.)

## K.3

## The Intel 80x86

### Introduction

MIPS was the vision of a single architect. The pieces of this architecture fit nicely together and the whole architecture can be described succinctly. Such is not the case of the 80x86: It is the product of several independent groups who evolved the architecture over 20 years, adding new features to the original instruction set as you might add clothing to a packed bag. Here are important 80x86 milestones:

- 1978—The Intel 8086 architecture was announced as an assembly language-compatible extension of the then-successful Intel 8080, an 8-bit microprocessor. The 8086 is a 16-bit architecture, with all internal registers 16 bits wide. Whereas the 8080 was a straightforward accumulator machine, the 8086 extended the architecture with additional registers. Because nearly every register has a dedicated use, the 8086 falls somewhere between an accumulator machine and a general-purpose register machine, and can fairly be called an *extended accumulator* machine.
- 1980—The Intel 8087 floating-point coprocessor is announced. This architecture extends the 8086 with about 60 floating-point instructions. Its architects rejected extended accumulators to go with a hybrid of stacks and registers,

essentially an *extended stack* architecture: A complete stack instruction set is supplemented by a limited set of register-memory instructions.

- 1982—The 80286 extended the 8086 architecture by increasing the address space to 24 bits, by creating an elaborate memory mapping and protection model, and by adding a few instructions to round out the instruction set and to manipulate the protection model. Because it was important to run 8086 programs without change, the 80286 offered a *real addressing mode* to make the machine look just like an 8086.
- 1985—The 80386 extended the 80286 architecture to 32 bits. In addition to a 32-bit architecture with 32-bit registers and a 32-bit address space, the 80386 added new addressing modes and additional operations. The added instructions make the 80386 nearly a general-purpose register machine. The 80386 also added paging support in addition to segmented addressing (see Chapter 2). Like the 80286, the 80386 has a mode to execute 8086 programs without change.

This history illustrates the impact of the “golden handcuffs” of compatibility on the 80x86, as the existing software base at each step was too important to jeopardize with significant architectural changes. Fortunately, the subsequent 80486 in 1989, Pentium in 1992, and P6 in 1995 were aimed at higher performance, with only four instructions added to the user-visible instruction set: three to help with multiprocessing plus a conditional move instruction.

Since 1997 Intel has added hundreds of instructions to support multimedia by operating on many narrower data types within a single clock (see Appendix A). These SIMD or vector instructions are primarily used in hand-coded libraries or drivers and rarely generated by compilers. The first extension, called MMX, appeared in 1997. It consists of 57 instructions that pack and unpack multiple bytes, 16-bit words, or 32-bit double words into 64-bit registers and performs shift, logical, and integer arithmetic on the narrow data items in parallel. It supports both saturating and nonsaturating arithmetic. MMX uses the registers comprising the floating-point stack and hence there is no new state for operating systems to save.

In 1999 Intel added another 70 instructions, labeled SSE, as part of Pentium III. The primary changes were to add eight separate registers, double their width to 128 bits, and add a single-precision floating-point data type. Hence, four 32-bit floating-point operations can be performed in parallel. To improve memory performance, SSE included cache prefetch instructions plus streaming store instructions that bypass the caches and write directly to memory.

In 2001, Intel added yet another 144 instructions, this time labeled SSE2. The new data type is double-precision arithmetic, which allows pairs of 64-bit floating-point operations in parallel. Almost all of these 144 instructions are versions of existing MMX and SSE instructions that operate on 64 bits of data in parallel. Not only does this change enable multimedia operations, but it also gives the compiler a different target for floating-point operations than the unique stack architecture. Compilers can choose to use the eight SSE registers as floating-point registers as found in the RISC machines. This change has boosted performance on the Pentium 4, the first microprocessor to include SSE2 instructions. At the time of

announcement, a 1.5 GHz Pentium 4 was 1.24 times faster than a 1 GHz Pentium III for SPECint2000(base), but it was 1.88 times faster for SPECfp2000(base).

In 2003 a company other than Intel enhanced the IA-32 architecture this time. AMD announced a set of architectural extensions to increase the address space for 32 to 64 bits. Similar to the transition from 16- to 32-bit address space in 1985 with the 80386, AMD64 widens all registers to 64 bits. It also increases the number of registers to sixteen and has 16 128-bit registers to support XMM, AMD's answer to SSE2. Rather than expand the instruction set, the primary change is adding a new mode called *long mode* that redefines the execution of all IA-32 instructions with 64-bit addresses. To address the larger number of registers, it adds a new prefix to instructions. AMD64 still has a 32-bit mode that is backwards compatible to the standard Intel instruction set, allowing a more graceful transition to 64-bit addressing than the HP/Intel Itanium. Intel later followed AMD's lead, making almost identical changes so that most software can run on either 64-bit address version of the 80x86 without change.

Whatever the artistic failures of the 80x86, keep in mind that there are more instances of this architectural family than of any other server or desktop processor in the world. Nevertheless, its checkered ancestry has led to an architecture that is difficult to explain and impossible to love.

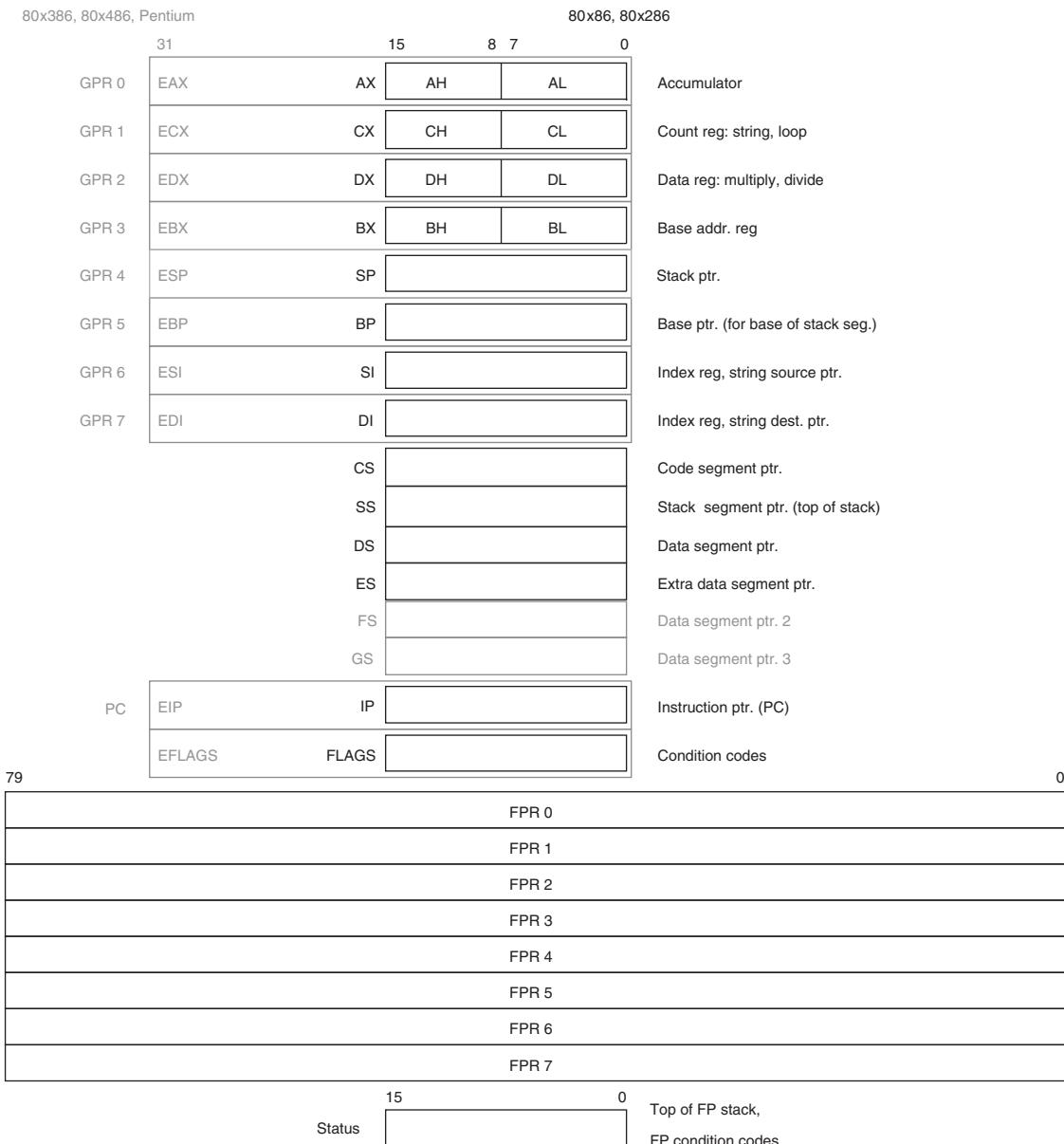
We start our explanation with the registers and addressing modes, move on to the integer operations, then cover the floating-point operations, and conclude with an examination of instruction encoding.

## 80x86 Registers and Data Addressing Modes

The evolution of the instruction set can be seen in the registers of the 80x86 (Figure K.30). Original registers are shown in black type, with the extensions of the 80386 shown in a lighter shade, a coloring scheme followed in subsequent figures. The 80386 basically extended all 16-bit registers (except the segment registers) to 32 bits, prefixing an "E" to their name to indicate the 32-bit version. The arithmetic, logical, and data transfer instructions are two-operand instructions that allow the combinations shown in Figure K.31.

To explain the addressing modes, we need to keep in mind whether we are talking about the 16-bit mode used by both the 8086 and 80286 or the 32-bit mode available on the 80386 and its successors. The seven data memory addressing modes supported are

- Absolute
- Register indirect
- Based
- Indexed
- Based indexed with displacement
- Based with scaled indexed
- Based with scaled indexed and displacement



**Figure K.30** The 80x86 has evolved over time, and so has its register set. The original set is shown in black and the extended set in gray. The 8086 divided the first four registers in half so that they could be used either as one 16-bit register or as two 8-bit registers. Starting with the 80386, the top eight registers were extended to 32 bits and could also be used as general-purpose registers. The floating-point registers on the bottom are 80 bits wide, and although they look like regular registers they are not. They implement a stack, with the top of stack pointed to by the status register. One operand must be the top of stack, and the other can be any of the other seven registers below the top of stack.

| Source/destination operand type | Second source operand |
|---------------------------------|-----------------------|
| Register                        | Register              |
| Register                        | Immediate             |
| Register                        | Memory                |
| Memory                          | Register              |
| Memory                          | Immediate             |

**Figure K.31 Instruction types for the arithmetic, logical, and data transfer instructions.** The 80x86 allows the combinations shown. The only restriction is the absence of a memory-memory mode. Immediates may be 8, 16, or 32 bits in length; a register is any one of the 14 major registers in Figure K.30 (not IP or FLAGS).

Displacements can be 8 or 32 bits in 32-bit mode, and 8 or 16 bits in 16-bit mode. If we count the size of the address as a separate addressing mode, the total is 11 addressing modes.

Although a memory operand can use any addressing mode, there are restrictions on what registers can be used in a mode. The section “80x86 Instruction Encoding” on page K-11 gives the full set of restrictions on registers, but the following description of addressing modes gives the basic register options:

- *Absolute*—With 16-bit or 32-bit displacement, depending on the mode.
- *Register indirect*—BX, SI, DI in 16-bit mode and EAX, ECX, EDX, EBX, ESI, and EDI in 32-bit mode.
- *Based mode with 8-bit or 16-bit/32-bit displacement*—BP, BX, SI, and DI in 16-bit mode and EAX, ECX, EDX, EBX, ESI, and EDI in 32-bit mode. The displacement is either 8 bits or the size of the address mode: 16 or 32 bits. (Intel gives two different names to this single addressing mode, *based* and *indexed*, but they are essentially identical and we combine them. This book uses indexed addressing to mean something different, explained next.)
- *Indexed*—The address is the sum of two registers. The allowable combinations are BX+SI, BX+DI, BP+SI, and BP+DI. This mode is called *based indexed* on the 8086. (The 32-bit mode uses a different addressing mode to get the same effect.)
- *Based indexed with 8- or 16-bit displacement*—The address is the sum of displacement and contents of two registers. The same restrictions on registers apply as in indexed mode.
- *Base plus scaled indexed*—This addressing mode and the next were added in the 80386 and are only available in 32-bit mode. The address calculation is

$$\text{Base register} + 2^{\text{Scale}} \times \text{Index} \times \text{register}$$

where *Scale* has the value 0, 1, 2, or 3; *Index register* can be any of the eight 32-bit general registers except ESP; and *Base register* can be any of the eight 32-bit general registers.

- *Base plus scaled index with 8- or 32-bit displacement*—The address is the sum of the displacement and the address calculated by the scaled mode immediately above. The same restrictions on registers apply.

The 80x86 uses Little Endian addressing.

Ideally, we would refer discussion of 80x86 logical and physical addresses to Chapter 2, but the segmented address space prevents us from hiding that information. Figure K.32 shows the memory mapping options on the generations of 80x86 machines; Chapter 2 describes the segmented protection scheme in greater detail.

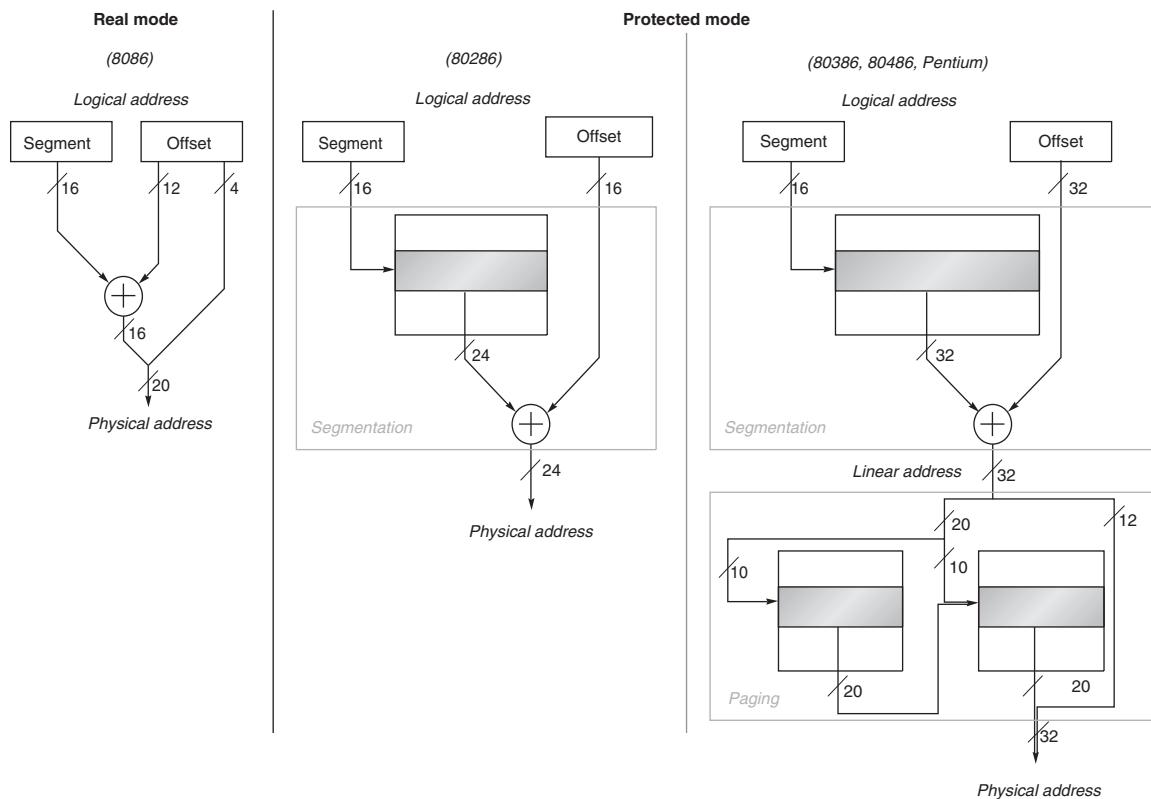
The assembly language programmer clearly must specify which segment register should be used with an address, no matter which address mode is used. To save space in the instructions, segment registers are selected automatically depending on which address register is used. The rules are simple: References to instructions (IP) use the code segment register (CS), references to the stack (BP or SP) use the stack segment register (SS), and the default segment register for the other registers is the data segment register (DS). The next section explains how they can be overridden.

## 80x86 Integer Operations

The 8086 provides support for both 8-bit (*byte*) and 16-bit (called *word*) data types. The data type distinctions apply to register operations as well as memory accesses. The 80386 adds 32-bit addresses and data, called *double words*. Almost every operation works on both 8-bit data and one longer data size. That size is determined by the mode and is either 16 or 32 bits.

Clearly some programs want to operate on data of all three sizes, so the 80x86 architects provide a convenient way to specify each version without expanding code size significantly. They decided that most programs would be dominated by either 16- or 32-bit data, and so it made sense to be able to set a default large size. This default size is set by a bit in the code segment register. To override the default size, an 8-bit *prefix* is attached to the instruction to tell the machine to use the other large size for this instruction.

The prefix solution was borrowed from the 8086, which allows multiple prefixes to modify instruction behavior. The three original prefixes override the default segment register, lock the bus so as to perform a semaphore (see Chapter 5), or repeat the following instruction until CX counts down to zero. This last prefix was intended to be paired with a byte move instruction to move a variable number of bytes. The 80386 also added a prefix to override the default address size.



**Figure K.32** The original segmented scheme of the 8086 is shown on the left. All 80x86 processors support this style of addressing, called *real mode*. It simply takes the contents of a segment register, shifts it left 4 bits, and adds it to the 16-bit offset, forming a 20-bit physical address. The 80286 (center) used the contents of the segment register to select a segment descriptor, which includes a 24-bit base address among other items. It is added to the 16-bit offset to form the 24-bit physical address. The 80386 and successors (right) expand this base address in the segment descriptor to 32 bits and also add an optional paging layer below segmentation. A 32-bit linear address is first formed from the segment and offset, and then this address is divided into two 10-bit fields and a 12-bit page offset. The first 10-bit field selects the entry in the first-level page table, and then this entry is used in combination with the second 10-bit field to access the second-level page table to select the upper 20 bits of the physical address. Prepending this 20-bit address to the final 12-bit field gives the 32-bit physical address. Paging can be turned off, redefining the 32-bit linear address as the physical address. Note that a “flat” 80x86 address space comes simply by loading the same value in all the segment registers; that is, it doesn’t matter which segment register is selected.

The 80x86 integer operations can be divided into four major classes:

1. Data movement instructions, including move, push, and pop
2. Arithmetic and logic instructions, including logical operations, test, shifts, and integer and decimal arithmetic operations
3. Control flow, including conditional branches and unconditional jumps, calls, and returns
4. String instructions, including string move and string compare

| Instruction     | Function                                                           |
|-----------------|--------------------------------------------------------------------|
| JE name         | if equal(CC) {IP←name}; IP-128 ≤ name ≤ IP+128                     |
| JMP name        | IP← name                                                           |
| CALLF name, seg | SP←SP-2; M[SS:SP]←IP+5; SP←SP-2;<br>M[SS:SP]←CS; IP← name; CS←seg; |
|                 | MOVW BX,[DI+45] BX← <sub>16</sub> M[DS:DI+45]                      |
| PUSH SI         | SP←SP-2; M[SS:SP]←SI                                               |
| POP DI          | DI←M[SS:SP]; SP←SP+2                                               |
| ADD AX,#6765    | AX←AX+6765                                                         |
| SHL BX,1        | BX←BX <sub>1..15</sub> ## 0                                        |
| TEST DX,#42     | Set CC flags with DX & 42                                          |
| MOVSB           | M[ES:DI]← <sub>8</sub> M[DS:SI]; DI←DI+1; SI←SI+1                  |

**Figure K.33 Some typical 80x86 instructions and their functions.** A list of frequent operations appears in Figure K.34. We use the abbreviation SR:X to indicate the formation of an address with segment register SR and offset X. This effective address corresponding to SR:X is (SR<<4)+X. The CALLF saves the IP of the next instruction and the current CS on the stack.

Figure K.33 shows some typical 80x86 instructions and their functions.

The data transfer, arithmetic, and logic instructions are unremarkable, except that the arithmetic and logic instruction operations allow the destination to be either a register or a memory location.

Control flow instructions must be able to address destinations in another segment. This is handled by having two types of control flow instructions: “near” for intrasegment (within a segment) and “far” for intersegment (between segments) transfers. In far jumps, which must be unconditional, two 16-bit quantities follow the opcode in 16-bit mode. One of these is used as the instruction pointer, while the other is loaded into CS and becomes the new code segment. In 32-bit mode the first field is expanded to 32 bits to match the 32-bit program counter (EIP).

Calls and returns work similarly—a far call pushes the return instruction pointer and return segment on the stack and loads both the instruction pointer and the code segment. A far return pops both the instruction pointer and the code segment from the stack. Programmers or compiler writers must be sure to always use the same type of call *and* return for a procedure—a near return does not work with a far call, and *vice versa*.

String instructions are part of the 8080 ancestry of the 80x86 and are not commonly executed in most programs.

Figure K.34 lists some of the integer 80x86 instructions. Many of the instructions are available in both byte and word formats.

| Instruction                | Meaning                                                                                     |
|----------------------------|---------------------------------------------------------------------------------------------|
| <b>Control</b>             | <b>Conditional and unconditional branches</b>                                               |
| JNZ, JZ                    | Jump if condition to IP + 8-bit offset; JNE (for JNZ) and JE (for JZ) are alternative names |
| JMP, JMPF                  | Unconditional jump—8- or 16-bit offset intrasegment (near) and intersegment (far) versions  |
| CALL, CALLF                | Subroutine call—16-bit offset; return address pushed; near and far versions                 |
| RET, RETF                  | Pops return address from stack and jumps to it; near and far versions                       |
| LOOP                       | Loop branch—decrement CX; jump to IP + 8-bit displacement if CX ≠ 0                         |
| <b>Data transfer</b>       | <b>Move data between registers or between register and memory</b>                           |
| MOV                        | Move between two registers or between register and memory                                   |
| PUSH                       | Push source operand on stack                                                                |
| POP                        | Pop operand from stack top to a register                                                    |
| LES                        | Load ES and one of the GPRs from memory                                                     |
| <b>Arithmetic/logical</b>  | <b>Arithmetic and logical operations using the data registers and memory</b>                |
| ADD                        | Add source to destination; register-memory format                                           |
| SUB                        | Subtract source from destination; register-memory format                                    |
| CMP                        | Compare source and destination; register-memory format                                      |
| SHL                        | Shift left                                                                                  |
| SHR                        | Shift logical right                                                                         |
| RCR                        | Rotate right with carry as fill                                                             |
| CBW                        | Convert byte in AL to word in AX                                                            |
| TEST                       | Logical AND of source and destination sets flags                                            |
| INC                        | Increment destination; register-memory format                                               |
| DEC                        | Decrement destination; register-memory format                                               |
| OR                         | Logical OR; register-memory format                                                          |
| XOR                        | Exclusive OR; register-memory format                                                        |
| <b>String instructions</b> | <b>Move between string operands; length given by a repeat prefix</b>                        |
| MOVS                       | Copies from string source to destination; may be repeated                                   |
| LODS                       | Loads a byte or word of a string into the A register                                        |

**Figure K.34 Some typical operations on the 80x86.** Many operations use register-memory format, where either the source or the destination may be memory and the other may be a register or immediate operand.

## 80x86 Floating-Point Operations

Intel provided a stack architecture with its floating-point instructions: loads push numbers onto the stack, operations find operands in the top two elements of the stacks, and stores can pop elements off the stack, just as the stack example in Figure A.31 on page A-4 suggests.

Intel supplemented this stack architecture with instructions and addressing modes that allow the architecture to have some of the benefits of a register-memory model. In addition to finding operands in the top two elements of the stack, one operand can be in memory or in one of the seven registers below the top of the stack.

This hybrid is still a restricted register-memory model, however, in that loads always move data to the top of the stack while incrementing the top of stack pointer and stores can only move the top of stack to memory. Intel uses the notation  $ST$  to indicate the top of stack, and  $ST(i)$  to represent the  $i$ th register below the top of stack.

One novel feature of this architecture is that the operands are wider in the register stack than they are stored in memory, and all operations are performed at this wide internal precision. Numbers are automatically converted to the internal 80-bit format on a load and converted back to the appropriate size on a store. Memory data can be 32-bit (single-precision) or 64-bit (double-precision) floating-point numbers, called *real* by Intel. The register-memory version of these instructions will then convert the memory operand to this Intel 80-bit format before performing the operation. The data transfer instructions also will automatically convert 16- and 32-bit integers to reals, and *vice versa*, for integer loads and stores.

The 80x86 floating-point operations can be divided into four major classes:

1. Data movement instructions, including load, load constant, and store
2. Arithmetic instructions, including add, subtract, multiply, divide, square root, and absolute value
3. Comparison, including instructions to send the result to the integer CPU so that it can branch
4. Transcendental instructions, including sine, cosine, log, and exponentiation

Figure K.35 shows some of the 60 floating-point operations. We use the curly brackets {} to show optional variations of the basic operations: {I} means there is an integer version of the instruction, {P} means this variation will pop one operand off the stack after the operation, and {R} means reverse the sense of the operands in this operation.

Not all combinations are provided. Hence,

$F\{I\}SUB\{R\}\{P\}$

represents these instructions found in the 80x86:

FSUB  
FISUB  
FSUBR  
FISUBR  
FSUBP  
FSUBRP

| Data transfer       | Arithmetic             | Compare        | Transcendental |
|---------------------|------------------------|----------------|----------------|
| F{I}LD mem/ST(i)    | F{I}ADD{P}mem/ST(i)    | F{I}COM{P}{P}  | FPATAN         |
| F{I}ST{P} mem/ST(i) | F{I}SUB{R}{P}mem/ST(i) | F{I}UCOM{P}{P} | F2XM1          |
| FLDPI               | F{I}MUL{P}mem/ST(i)    | FSTSW AX/mem   | FCOS           |
| FLD1                | F{I}DIV{R}{P}mem/ST(i) |                | FPTAN          |
| FLDZ                | FSQRT                  |                | FPREM          |
|                     | FABS                   |                | FSIN           |
|                     | FRNDINT                |                | FYL2X          |

**Figure K.35** The floating-point instructions of the 80x86. The first column shows the data transfer instructions, which move data to memory or to one of the registers below the top of the stack. The last three operations push constants on the stack: pi, 1.0, and 0.0. The second column contains the arithmetic operations described above. Note that the last three operate only on the top of stack. The third column is the compare instructions. Since there are no special floating-point branch instructions, the result of the compare must be transferred to the integer CPU via the FSTSW instruction, either into the AX register or into memory, followed by an SAHF instruction to set the condition codes. The floating-point comparison can then be tested using integer branch instructions. The final column gives the higher-level floating-point operations.

There are no pop or reverse pop versions of the integer subtract instructions.

Note that we get even more combinations when including the operand modes for these operations. The floating-point add has these options, ignoring the integer and pop versions of the instruction:

|      |           |                                                                                                              |
|------|-----------|--------------------------------------------------------------------------------------------------------------|
| FADD |           | Both operands are in the stack, and the result replaces the top of stack.                                    |
| FADD | ST(i)     | One source operand is <i>i</i> th register below the top of stack, and the result replaces the top of stack. |
| FADD | ST(i), ST | One source operand is the top of stack, and the result replaces <i>i</i> th register below the top of stack. |
| FADD | mem32     | One source operand is a 32-bit location in memory, and the result replaces the top of stack.                 |
| FADD | mem64     | One source operand is a 64-bit location in memory, and the result replaces the top of stack.                 |

As mentioned earlier SSE2 presents a model of IEEE floating-point registers.

## 80x86 Instruction Encoding

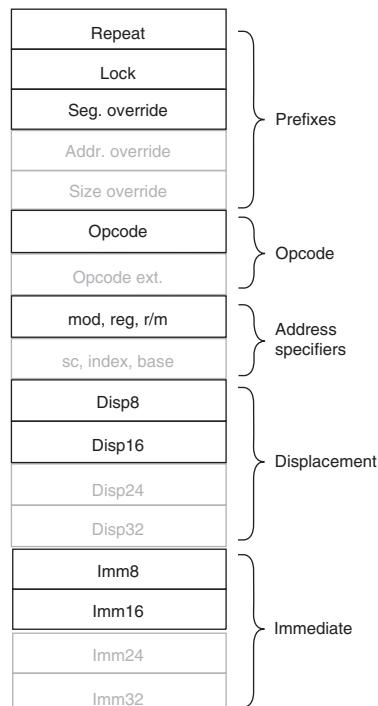
Saving the worst for last, the encoding of instructions in the 8086 is complex, with many different instruction formats. Instructions may vary from 1 byte, when there are no operands, to up to 6 bytes, when the instruction contains a 16-bit immediate

and uses 16-bit displacement addressing. Prefix instructions increase 8086 instruction length beyond the obvious sizes.

The 80386 additions expand the instruction size even further, as Figure K.36 shows. Both the displacement and immediate fields can be 32 bits long, two more prefixes are possible, the opcode can be 16 bits long, and the scaled index mode specifier adds another 8 bits. The maximum possible 80386 instruction is 17 bytes long.

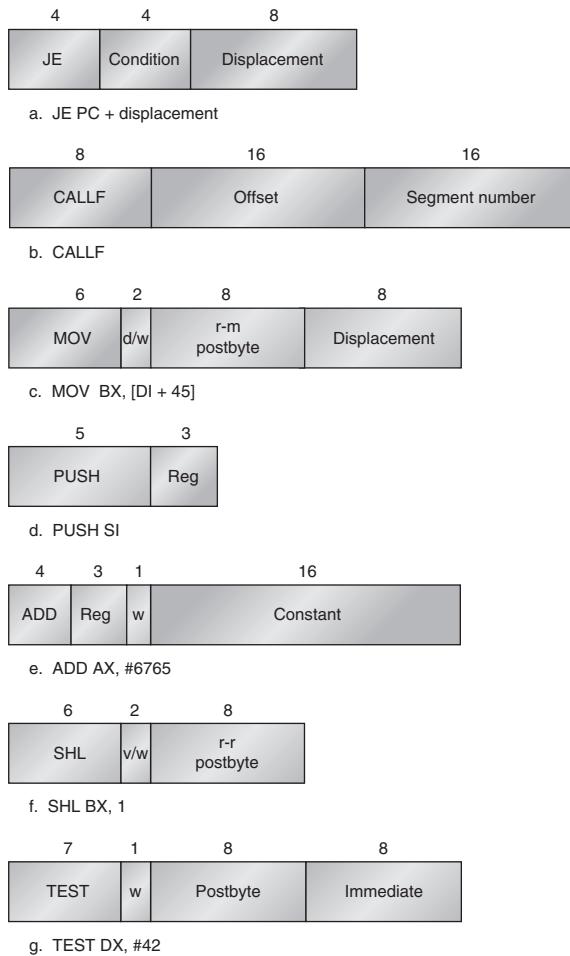
Figure K.37 shows the instruction format for several of the example instructions in Figure K.33. The opcode byte usually contains a bit saying whether the operand is a byte wide or the larger size, 16 bits or 32 bits depending on the mode. For some instructions, the opcode may include the addressing mode and the register; this is true in many instructions that have the form register  $\leftarrow$  register op immediate. Other instructions use a “postbyte” or extra opcode byte, labeled “mod, reg, r/m” in Figure K.36, which contains the addressing mode information. This postbyte is used for many of the instructions that address memory. The based with scaled index uses a second postbyte, labeled “sc, index, base” in Figure K.36.

The floating-point instructions are encoded in the escape opcode of the 8086 and the postbyte address specifier. The memory operations reserve 2 bits to decide




---

**Figure K.36** The instruction format of the 8086 (black type) and the extensions for the 80386 (shaded type). Every field is optional except the opcode.



**Figure K.37** Typical 8086 instruction formats. The encoding of the postbyte is shown in Figure K.38. Many instructions contain the 1-bit field *w*, which says whether the operation is a byte or a word. Fields of the form *v/w* or *d/w* are a *d*-field or *v*-field followed by the *w*-field. The *d*-field in *MOV* is used in instructions that may move to or from memory and shows the direction of the move. The field *v* in the *SHL* instruction indicates a variable-length shift; variable-length shifts use a register to hold the shift count. The *ADD* instruction shows a typical optimized short encoding usable only when the first operand is *AX*. Overall instructions may vary from 1 to 6 bytes in length.

whether the operand is a 32- or 64-bit real or a 16- or 32-bit integer. Those same 2 bits are used in versions that do not access memory to decide whether the stack should be popped after the operation and whether the top of stack or a lower register should get the result.

Alas, you cannot separate the restrictions on registers from the encoding of the addressing modes in the 80x86. Hence, Figures K.38 and K.39 show the encoding of the two postbyte address specifiers for both 16- and 32-bit mode.

|     |       | w = 1 |     | mod = 0 |             | mod = 1  |                       | mod = 2   |              | mod = 3    |         |
|-----|-------|-------|-----|---------|-------------|----------|-----------------------|-----------|--------------|------------|---------|
| reg | w = 0 | 16b   | 32b | r/m     | 16b         | 32b      | 16b                   | 32b       | 16b          | 32b        | mod = 3 |
| 0   | A L   | A X   | EAX | 0       | addr=BX+SI  | =EAX     | same                  | same      | same         | same       | same    |
| 1   | C L   | C X   | ECX | 1       | addr=BX+DI  | =ECX     | addr as               | addr as   | addr as      | addr as    | as      |
| 2   | D L   | D X   | EDX | 2       | addr=BP+SI  | =ED X    | mod= 0                | mod= 0    | mod= 0       | mod= 0     | reg     |
| 3   | B L   | B X   | EBX | 3       | addr=BP+SI  | =EB X    | + disp 8              | + disp 8  | + disp1 6    | + disp3 2  | field   |
| 4   | A H   | SP    | ESP | 4       | addr=SI     | = (si) b | SI+disp16 (sib)+disp8 | SI+disp8  | (sib)+disp32 | "          |         |
| 5   | C H   | B P   | EBP | 5       | addr=DI     | =disp32  | DI+disp8              | EBP+disp8 | DI+disp16    | EBP+disp32 | "       |
| 6   | D H   | SI    | ESI | 6       | addr=disp16 | =ESI     | BP+disp8              | ESI+disp8 | BP+disp16    | ESI+disp32 | "       |
| 7   | B H   | D I   | EDI | 7       | addr=BX     | =ED I    | BX+disp8              | EDI+disp8 | BX+disp16    | EDI+disp32 | "       |

**Figure K.38** The encoding of the first address specifier of the 80x86, mod, reg, r/m. The first four columns show the encoding of the 3-bit reg field, which depends on the w bit from the opcode and whether the machine is in 16- or 32-bit mode. The remaining columns explain the mod and r/m fields. The meaning of the 3-bit r/m field depends on the value in the 2-bit mod field and the address size. Basically, the registers used in the address calculation are listed in the sixth and seventh columns, under mod = 0, with mod = 1 adding an 8-bit displacement and mod = 2 adding a 16- or 32-bit displacement, depending on the address mode. The exceptions are r/m = 6 when mod = 1 or mod = 2 in 16-bit mode selects BP plus the displacement; r/m = 5 when mod = 1 or mod = 2 in 32-bit mode selects EBP plus displacement; and r/m = 4 in 32-bit mode when mod  $\neq$  3 (sib) means use the scaled index mode shown in Figure K.39. When mod = 3, the r/m field indicates a register, using the same encoding as the reg field combined with the w bit.

| Index | Base     |
|-------|----------|
| 0     | EAX      |
| 1     | ECX      |
| 2     | EDX      |
| 3     | EBX      |
| 4     | No index |
| 5     | EBP      |
| 6     | ESI      |
| 7     | EDI      |

**Figure K.39** Based plus scaled index mode address specifier found in the 80386. This mode is indicated by the (sib) notation in Figure K.38. Note that this mode expands the list of registers to be used in other modes: Register indirect using ESP comes from Scale = 0, Index = 4, and Base = 4, and base displacement with EBP comes from Scale = 0, Index = 5, and mod = 0. The two-bit scale field is used in this formula of the effective address: Base register +  $2^{\text{Scale}} \times$  Index register.

## Putting It All Together: Measurements of Instruction Set Usage

In this section, we present detailed measurements for the 80x86 and then compare the measurements to MIPS for the same programs. To facilitate comparisons among dynamic instruction set measurements, we use a subset of the SPEC92 programs. The 80x86 results were taken in 1994 using the Sun Solaris FORTRAN and C compilers V2.0 and executed in 32-bit mode. These compilers were comparable in quality to the compilers used for MIPS.

Remember that these measurements depend on the benchmarks chosen and the compiler technology used. Although we feel that the measurements in this section are reasonably indicative of the usage of these architectures, other programs may behave differently from any of the benchmarks here, and different compilers may yield different results. In doing a real instruction set study, the architect would want to have a much larger set of benchmarks, spanning as wide an application range as possible, and consider the operating system and its usage of the instruction set. Single-user benchmarks like those measured here do not necessarily behave in the same fashion as the operating system.

We start with an evaluation of the features of the 80x86 in isolation, and later compare instruction counts with those of DLX.

### *Measurements of 80x86 Operand Addressing*

We start with addressing modes. Figure K.40 shows the distribution of the operand types in the 80x86. These measurements cover the “second” operand of the operation; for example,

```
mov EAX, [45]
```

counts as a single memory operand. If the types of the first operand were counted, the percentage of register usage would increase by about a factor of 1.5.

The 80x86 memory operands are divided into their respective addressing modes in Figure K.41. Probably the biggest surprise is the popularity of the

---

|           | Integer average | FP average |
|-----------|-----------------|------------|
| Register  | 45%             | 22%        |
| Immediate | 16%             | 6%         |
| Memory    | 39%             | 72%        |

---

**Figure K.40** Operand type distribution for the average of five SPECint92 programs (compress, eqntott, espresso, gcc, li) and the average of five SPECfp92 programs (doduc, ear, hydro2d, mdljdp2, su2cor).

| Addressing mode                      | Integer average | FP average |
|--------------------------------------|-----------------|------------|
| Register indirect                    | 13%             | 3%         |
| Base + 8-bit disp.                   | 31%             | 15%        |
| Base + 32-bit disp.                  | 9%              | 25%        |
| Indexed                              | 0%              | 0%         |
| Based indexed + 8-bit disp.          | 0%              | 0%         |
| Based indexed + 32-bit disp.         | 0%              | 1%         |
| Base + scaled indexed                | 22%             | 7%         |
| Base + scaled indexed + 8-bit disp.  | 0%              | 8%         |
| Base + scaled indexed + 32-bit disp. | 4%              | 4%         |
| 32-bit direct                        | 20%             | 37%        |

**Figure K.41 Operand addressing mode distribution by program.** This chart does not include addressing modes used by branches or control instructions.

addressing modes added by the 80386, the last four rows of the figure. They account for about half of all the memory accesses. Another surprise is the popularity of direct addressing. On most other machines, the equivalent of the direct addressing mode is rare. Perhaps the segmented address space of the 80x86 makes direct addressing more useful, since the address is relative to a base address from the segment register.

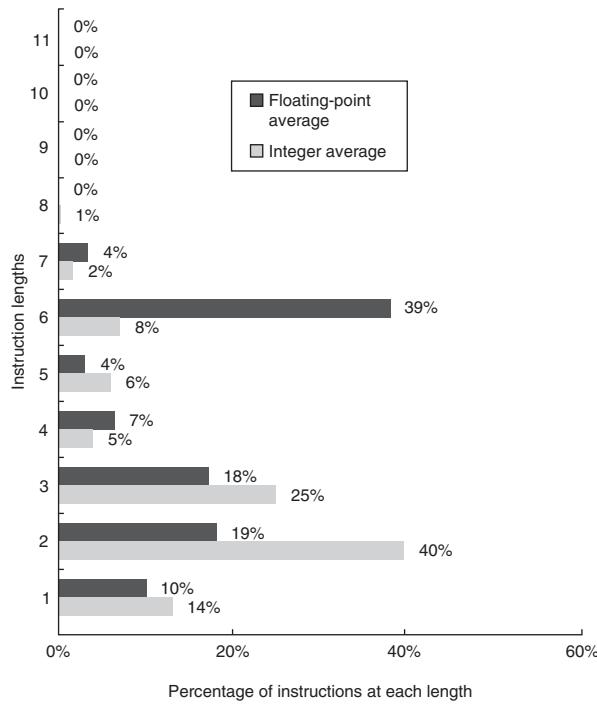
These addressing modes largely determine the size of the Intel instructions. Figure K.42 shows the distribution of instruction sizes. The average number of bytes per instruction for integer programs is 2.8, with a standard deviation of 1.5, and 4.1 with a standard deviation of 1.9 for floating-point programs. The difference in length arises partly from the differences in the addressing modes: Integer programs rely more on the shorter register indirect and 8-bit displacement addressing modes, while floating-point programs more frequently use the 80386 addressing modes with the longer 32-bit displacements.

Given that the floating-point instructions have aspects of both stacks and registers, how are they used? Figure K.43 shows that, at least for the compilers used in this measurement, the stack model of execution is rarely followed. (See Section L.3 for a historical explanation of this observation.)

Finally, Figures K.44 and K.45 show the instruction mixes for 10 SPEC92 programs.

#### *Comparative Operation Measurements*

Figures K.46 and K.47 show the number of instructions executed for each of the 10 programs on the 80x86 and the ratio of instruction execution compared with that



**Figure K.42** Averages of the histograms of 80x86 instruction lengths for five SPECint92 programs and for five SPECfp92 programs, all running in 32-bit mode.

| Option                              | doduc | ear   | hydro2d | mdljdp2 | su2cor | FP average |
|-------------------------------------|-------|-------|---------|---------|--------|------------|
| Stack (2nd operand ST (1))          | 1.1%  | 0.0%  | 0.0%    | 0.2%    | 0.6%   | 0.4%       |
| Register (2nd operand ST(i), i > 1) | 17.3% | 63.4% | 14.2%   | 7.1%    | 30.7%  | 26.5%      |
| Memory                              | 81.6% | 36.6% | 85.8%   | 92.7%   | 68.7%  | 73.1%      |

**Figure K.43** The percentage of instructions for the floating-point operations (add, sub, mul, div) that use each of the three options for specifying a floating-point operand on the 80x86. The three options are (1) the strict stack model of implicit operands on the stack, (2) register version naming an explicit operand that is not one of the top two elements of the stack, and (3) memory operand.

for DLX: Numbers less than 1.0 mean that the 80x86 executes fewer instructions than DLX. The instruction count is surprisingly close to DLX for many integer programs, as you would expect a load-store instruction set architecture like DLX to execute more instructions than a register-memory architecture like the 80x86. The floating-point programs always have higher counts for the 80x86,

| <b>Instruction</b>       | <b>doduc</b> | <b>ear</b> | <b>hydro2d</b> | <b>mdljdp2</b> | <b>su2cor</b> | <b>FP average</b> |
|--------------------------|--------------|------------|----------------|----------------|---------------|-------------------|
| Load                     | 8.9%         | 6.5%       | 18.0%          | 27.6%          | 27.6%         | 20%               |
| Store                    | 12.4%        | 3.1%       | 11.5%          | 7.8%           | 7.8%          | 8%                |
| Add                      | 5.4%         | 6.6%       | 14.6%          | 8.8%           | 8.8%          | 10%               |
| Sub                      | 1.0%         | 2.4%       | 3.3%           | 2.4%           | 2.4%          | 3%                |
| Mul                      |              |            |                |                |               | 0%                |
| Div                      |              |            |                |                |               | 0%                |
| Compare                  | 1.8%         | 5.1%       | 0.8%           | 1.0%           | 1.0%          | 2%                |
| Mov reg-reg              | 3.2%         | 0.1%       | 1.8%           | 2.3%           | 2.3%          | 2%                |
| Load imm                 | 0.4%         | 1.5%       |                |                |               | 0%                |
| Cond. branch             | 5.4%         | 8.2%       | 5.1%           | 2.7%           | 2.7%          | 5%                |
| Uncond branch            | 0.8%         | 0.4%       | 1.3%           | 0.3%           | 0.3%          | 1%                |
| Call                     | 0.5%         | 1.6%       |                | 0.1%           | 0.1%          | 0%                |
| Return, jmp indirect     | 0.5%         | 1.6%       |                | 0.1%           | 0.1%          | 0%                |
| Shift                    | 1.1%         |            | 4.5%           | 2.5%           | 2.5%          | 2%                |
| AND                      | 0.8%         | 0.8%       | 0.7%           | 1.3%           | 1.3%          | 1%                |
| OR                       | 0.1%         |            |                | 0.1%           | 0.1%          | 0%                |
| Other (XOR, not, . . .)  |              |            |                |                |               | 0%                |
| Load FP                  | 14.1%        | 22.5%      | 9.1%           | 12.6%          | 12.6%         | 14%               |
| Store FP                 | 8.6%         | 11.4%      | 4.1%           | 6.6%           | 6.6%          | 7%                |
| Add FP                   | 5.8%         | 6.1%       | 1.4%           | 6.6%           | 6.6%          | 5%                |
| Sub FP                   | 2.2%         | 2.7%       | 3.1%           | 2.9%           | 2.9%          | 3%                |
| Mul FP                   | 8.9%         | 8.0%       | 4.1%           | 12.0%          | 12.0%         | 9%                |
| Div FP                   | 2.1%         |            | 0.8%           | 0.2%           | 0.2%          | 0%                |
| Compare FP               | 9.4%         | 6.9%       | 10.8%          | 0.5%           | 0.5%          | 5%                |
| Mov reg-reg FP           | 2.5%         | 0.8%       | 0.3%           | 0.8%           | 0.8%          | 1%                |
| Other (abs, sqrt, . . .) | 3.9%         | 3.8%       | 4.1%           | 0.8%           | 0.8%          | 2%                |

**Figure K.44** 80x86 instruction mix for five SPECfp92 programs.

presumably due to the lack of floating-point registers and the use of a stack architecture.

Another question is the total amount of data traffic for the 80x86 versus DLX, since the 80x86 can specify memory operands as part of operations while DLX can only access via loads and stores. Figures K.46 and K.47 also show the data reads, data writes, and data read-modify-writes for these 10 programs. The total

| <b>Instruction</b>       | <b>compress</b> | <b>eqntott</b> | <b>espresso</b> | <b>gcc (cc1)</b> | <b>li</b> | <b>Int. average</b> |
|--------------------------|-----------------|----------------|-----------------|------------------|-----------|---------------------|
| Load                     | 20.8%           | 18.5%          | 21.9%           | 24.9%            | 23.3%     | 22%                 |
| Store                    | 13.8%           | 3.2%           | 8.3%            | 16.6%            | 18.7%     | 12%                 |
| Add                      | 10.3%           | 8.8%           | 8.15%           | 7.6%             | 6.1%      | 8%                  |
| Sub                      | 7.0%            | 10.6%          | 3.5%            | 2.9%             | 3.6%      | 5%                  |
| Mul                      |                 |                |                 | 0.1%             |           | 0%                  |
| Div                      |                 |                |                 |                  |           | 0%                  |
| Compare                  | 8.2%            | 27.7%          | 15.3%           | 13.5%            | 7.7%      | 16%                 |
| Mov reg-reg              | 7.9%            | 0.6%           | 5.0%            | 4.2%             | 7.8%      | 4%                  |
| Load imm                 | 0.5%            | 0.2%           | 0.6%            | 0.4%             |           | 0%                  |
| Cond. branch             | 15.5%           | 28.6%          | 18.9%           | 17.4%            | 15.4%     | 20%                 |
| Uncond. branch           | 1.2%            | 0.2%           | 0.9%            | 2.2%             | 2.2%      | 1%                  |
| Call                     | 0.5%            | 0.4%           | 0.7%            | 1.5%             | 3.2%      | 1%                  |
| Return, jmp indirect     | 0.5%            | 0.4%           | 0.7%            | 1.5%             | 3.2%      | 1%                  |
| Shift                    | 3.8%            |                | 2.5%            | 1.7%             |           | 1%                  |
| AND                      | 8.4%            | 1.0%           | 8.7%            | 4.5%             | 8.4%      | 6%                  |
| OR                       | 0.6%            |                | 2.7%            | 0.4%             | 0.4%      | 1%                  |
| Other (XOR, not, . . .)  | 0.9%            |                | 2.2%            | 0.1%             |           | 1%                  |
| Load FP                  |                 |                |                 |                  |           | 0%                  |
| Store FP                 |                 |                |                 |                  |           | 0%                  |
| Add FP                   |                 |                |                 |                  |           | 0%                  |
| Sub FP                   |                 |                |                 |                  |           | 0%                  |
| Mul FP                   |                 |                |                 |                  |           | 0%                  |
| Div FP                   |                 |                |                 |                  |           | 0%                  |
| Compare FP               |                 |                |                 |                  |           | 0%                  |
| Mov reg-reg FP           |                 |                |                 |                  |           | 0%                  |
| Other (abs, sqrt, . . .) |                 |                |                 |                  |           | 0%                  |

**Figure K.45** 80x86 instruction mix for five SPECint92 programs.

accesses ratio to DLX of each memory access type is shown in the bottom rows, with the read-modify-write counting as one read and one write. The 80x86 performs about two to four times as many data accesses as DLX for floating-point programs, and 1.25 times as many for integer programs. Finally, Figure K.48 shows the percentage of instructions in each category for 80x86 and DLX.

|                                             | <b>compress</b> | <b>eqntott</b> | <b>espresso</b> | <b>gcc (cc1)</b> | <b>li</b> | <b>Int. avg.</b> |
|---------------------------------------------|-----------------|----------------|-----------------|------------------|-----------|------------------|
| Instructions executed on 80x86 (millions)   | 2226            | 1203           | 2216            | 3770             | 5020      |                  |
| Instructions executed ratio to DLX          | 0.61            | 1.74           | 0.85            | 0.96             | 0.98      | 1.03             |
| Data reads on 80x86 (millions)              | 589             | 229            | 622             | 1079             | 1459      |                  |
| Data writes on 80x86 (millions)             | 311             | 39             | 191             | 661              | 981       |                  |
| Data read-modify-writes on 80x86 (millions) | 26              | 1              | 129             | 48               | 48        |                  |
| Total data reads on 80x86 (millions)        | 615             | 230            | 751             | 1127             | 1507      |                  |
| Data read ratio to DLX                      | 0.85            | 1.09           | 1.38            | 1.25             | 0.94      | 1.10             |
| Total data writes on 80x86 (millions)       | 338             | 40             | 319             | 709              | 1029      |                  |
| Data write ratio to DLX                     | 1.67            | 9.26           | 2.39            | 1.25             | 1.20      | 3.15             |
| Total data accesses on 80x86 (millions)     | 953             | 269            | 1070            | 1836             | 2536      |                  |
| Data access ratio to DLX                    | 1.03            | 1.25           | 1.58            | 1.25             | 1.03      | 1.23             |

**Figure K.46 Instructions executed and data accesses on 80x86 and ratios compared to DLX for five SPECint92 programs.**

|                                             | <b>doduc</b> | <b>ear</b> | <b>hydro2d</b> | <b>mdljdp2</b> | <b>su2cor</b> | <b>FP average</b> |
|---------------------------------------------|--------------|------------|----------------|----------------|---------------|-------------------|
| Instructions executed on 80x86 (millions)   | 1223         | 15,220     | 13,342         | 6197           | 6197          |                   |
| Instructions executed ratio to DLX          | 1.19         | 1.19       | 2.53           | 2.09           | 1.62          | 1.73              |
| Data reads on 80x86 (millions)              | 515          | 6007       | 5501           | 3696           | 3643          |                   |
| Data writes on 80x86 (millions)             | 260          | 2205       | 2085           | 892            | 892           |                   |
| Data read-modify-writes on 80x86 (millions) | 1            | 0          | 189            | 124            | 124           |                   |
| Total data reads on 80x86 (millions)        | 517          | 6007       | 5690           | 3820           | 3767          |                   |
| Data read ratio to DLX                      | 2.04         | 2.36       | 4.48           | 4.77           | 3.91          | 3.51              |
| Total data writes on 80x86 (millions)       | 261          | 2205       | 2274           | 1015           | 1015          |                   |
| Data write ratio to DLX                     | 3.68         | 33.25      | 38.74          | 16.74          | 9.35          | 20.35             |
| Total data accesses on 80x86 (millions)     | 778          | 8212       | 7965           | 4835           | 4782          |                   |
| Data access ratio to DLX                    | 2.40         | 3.14       | 5.99           | 5.73           | 4.47          | 4.35              |

**Figure K.47 Instructions executed and data accesses for five SPECfp92 programs on 80x86 and ratio to DLX.**

## Concluding Remarks

*Beauty is in the eye of the beholder.*

### Old Adage

As we have seen, “orthogonal” is not a term found in the Intel architectural dictionary. To fully understand which registers and which addressing modes are available, you need to see the encoding of all addressing modes and sometimes the encoding of the instructions.

| Category                 | Integer average |     | FP average |     |
|--------------------------|-----------------|-----|------------|-----|
|                          | x86             | DLX | x86        | DLX |
| Total data transfer      | 34%             | 36% | 28%        | 2%  |
| Total integer arithmetic | 34%             | 31% | 16%        | 12% |
| Total control            | 24%             | 20% | 6%         | 10% |
| Total logical            | 8%              | 13% | 3%         | 2%  |
| Total FP data transfer   | 0%              | 0%  | 22%        | 33% |
| Total FP arithmetic      | 0%              | 0%  | 25%        | 41% |

**Figure K.48** Percentage of instructions executed by category for 80x86 and DLX for the averages of five SPECint92 and SPECfp92 programs of Figures K.46 and K.47.

Some argue that the inelegance of the 80x86 instruction set is unavoidable, the price that must be paid for rampant success by any architecture. We reject that notion. Obviously, no successful architecture can jettison features that were added in previous implementations, and over time some features may be seen as undesirable. The awkwardness of the 80x86 began at its core with the 8086 instruction set and was exacerbated by the architecturally inconsistent expansions of the 8087, 80286, and 80386.

A counterexample is the IBM 360/370 architecture, which is much older than the 80x86. It dominates the mainframe market just as the 80x86 dominates the PC market. Due undoubtedly to a better base and more compatible enhancements, this instruction set makes much more sense than the 80x86 more than 30 years after its first implementation.

For better or worse, Intel had a 16-bit microprocessor years before its competitors' more elegant architectures, and this head start led to the selection of the 8086 as the CPU for the IBM PC. What it lacks in style is made up in quantity, making the 80x86 beautiful from the right perspective.

The saving grace of the 80x86 is that its architectural components are not too difficult to implement, as Intel has demonstrated by rapidly improving performance of integer programs since 1978. High floating-point performance is a larger challenge in this architecture.

## K.4

### The VAX Architecture

*VAX: the most successful minicomputer design in industry history . . . the VAX was probably the hacker's favorite machine . . . . Especially noted for its large, assembler-programmer-friendly instruction set—an asset that became a liability after the RISC revolution.*

**Eric Raymond**  
*The New Hacker's Dictionary* (1991)

## Introduction

To enhance your understanding of instruction set architectures, we chose the VAX as the representative *Complex Instruction Set Computer* (CISC) because it is so different from MIPS and yet still easy to understand. By seeing two such divergent styles, we are confident that you will be able to learn other instruction sets on your own.

At the time the VAX was designed, the prevailing philosophy was to create instruction sets that were close to programming languages in order to simplify compilers. For example, because programming languages had loops, instruction sets should have loop instructions. As VAX architect William Strecker said (“VAX-11/780—A Virtual Address Extension to the PDP-11 Family,” *AFIPS Proc.*, National Computer Conference, 1978):

A major goal of the VAX-11 instruction set was to provide for effective compiler generated code. Four decisions helped to realize this goal: 1) A very regular and consistent treatment of operators . . . . 2) An avoidance of instructions unlikely to be generated by a compiler . . . . 3) Inclusions of several forms of common operators . . . . 4) Replacement of common instruction sequences with single instructions . . . . Examples include procedure calling, multiway branching, loop control, and array subscript calculation.

Recall that DRAMs of the mid-1970s contained less than 1/1000th the capacity of today’s DRAMs, so code space was also critical. Hence, another prevailing philosophy was to minimize code size, which is de-emphasized in fixed-length instruction sets like MIPS. For example, MIPS address fields always use 16 bits, even when the address is very small. In contrast, the VAX allows instructions to be a variable number of bytes, so there is little wasted space in address fields.

Whole books have been written just about the VAX, so this VAX extension cannot be exhaustive. Hence, the following sections describe only a few of its addressing modes and instructions. To show the VAX instructions in action, later sections show VAX assembly code for two C procedures. The general style will be to contrast these instructions with the MIPS code that you are already familiar with.

The differing goals for VAX and MIPS have led to very different architectures. The VAX goals, simple compilers and code density, led to the powerful addressing modes, powerful instructions, and efficient instruction encoding. The MIPS goals were high performance via pipelining, ease of hardware implementation, and compatibility with highly optimizing compilers. The MIPS goals led to simple instructions, simple addressing modes, fixed-length instruction formats, and a large number of registers.

## VAX Operands and Addressing Modes

The VAX is a 32-bit architecture, with 32-bit-wide addresses and 32-bit-wide registers. Yet, the VAX supports many other data sizes and types, as Figure K.49 shows. Unfortunately, VAX uses the name “word” to refer to 16-bit quantities; in this text, a word means 32 bits. Figure K.49 shows the conversion between

| Bits | Data type        | MIPS name        | VAX name                 |
|------|------------------|------------------|--------------------------|
| 8    | Integer          | Byte             | Byte                     |
| 16   | Integer          | Half word        | Word                     |
| 32   | Integer          | Word             | Long word                |
| 32   | Floating point   | Single precision | F_floating               |
| 64   | Integer          | Double word      | Quad word                |
| 64   | Floating point   | Double precision | D_floating or G_floating |
| 8n   | Character string | Character        | Character                |

**Figure K.49** VAX data types, their lengths, and names. The first letter of the VAX type (b, w, l, f, q, d, g, c) is often used to complete an instruction name. Examples of move instructions include `movb`, `movw`, `movl`, `movf`, `movq`, `movd`, `movg`, and `movc3`. Each move instruction transfers an operand of the data type indicated by the letter following `mov`.

the MIPS data type names and the VAX names. Be careful when reading about VAX instructions, as they refer to the names of the VAX data types.

The VAX provides sixteen 32-bit registers. The VAX assembler uses the notation `r0`, `r1`, . . . , `r15` to refer to these registers, and we will stick to that notation. Alas, 4 of these 16 registers are effectively claimed by the instruction set architecture. For example, `r14` is the stack pointer (`sp`) and `r15` is the program counter (`pc`). Hence, `r15` cannot be used as a general-purpose register, and using `r14` is very difficult because it interferes with instructions that manipulate the stack. The other dedicated registers are `r12`, used as the argument pointer (`ap`), and `r13`, used as the frame pointer (`fp`); their purpose will become clear later. (Like MIPS, the VAX assembler accepts either the register number or the register name.)

VAX addressing modes include those discussed in Appendix A, which has all the MIPS addressing modes: *register*, *displacement*, *immediate*, and *PC-relative*. Moreover, all these modes can be used for jump addresses or for data addresses.

But that's not all the addressing modes. To reduce code size, the VAX has three lengths of addresses for displacement addressing: 8-bit, 16-bit, and 32-bit addresses called, respectively, *byte displacement*, *word displacement*, and *long displacement* addressing. Thus, an address can be not only as small as possible but also as large as necessary; large addresses need not be split, so there is no equivalent to the MIPS `lui` instruction (see Figure A.24 on page A-37).

Those are still not all the VAX addressing modes. Several have a *deferred* option, meaning that the object addressed is only the *address* of the real object, requiring another memory access to get the operand. This addressing mode is called *indirect addressing* in other machines. Thus, *register deferred*, *autoincrement deferred*, and *byte/word/long displacement deferred* are other addressing modes to choose from. For example, using the notation of the VAX assembler,

$r1$  means the operand is register 1 and  $(r1)$  means the operand is the location in memory pointed to by  $r1$ .

There is yet another addressing mode. *Indexed addressing* automatically converts the value in an index operand to the proper byte address to add to the rest of the address. For a 32-bit word, we needed to multiply the index of a 4-byte quantity by 4 before adding it to a base address. Indexed addressing, called *scaled addressing* on some computers, automatically multiplies the index of a 4-byte quantity by 4 as part of the address calculation.

To cope with such a plethora of addressing options, the VAX architecture separates the specification of the addressing mode from the specification of the operation. Hence, the opcode supplies the operation and the number of operands, and each operand has its own addressing mode specifier. Figure K.50 shows the name, assembler notation, example, meaning, and length of the address specifier.

The VAX style of addressing means that an operation doesn't know where its operands come from; a VAX add instruction can have three operands in registers, three operands in memory, or any combination of registers and memory operands.

| Addressing mode name                 | Syntax             | Example  | Meaning                                                  | Length of address specifier in bytes |
|--------------------------------------|--------------------|----------|----------------------------------------------------------|--------------------------------------|
| Literal                              | #value             | #-1      | -1                                                       | 1 (6-bit signed value)               |
| Immediate                            | #value             | #100     | 100                                                      | 1 + length of the immediate          |
| Register                             | rn                 | r3       | r3                                                       | 1                                    |
| Register deferred                    | (rn)               | (r3)     | Memory[r3]                                               | 1                                    |
| Byte/word/long displacement          | Displacement (rn)  | 100(r3)  | Memory[r3 + 100]                                         | 1 + length of the displacement       |
| Byte/word/long displacement deferred | @displacement (rn) | @100(r3) | Memory[Memory[r3 + 100]]                                 | 1 + length of the displacement       |
| Indexed (scaled)                     | Base mode [rx]     | (r3)[r4] | Memory[r3 + r4 × d]<br>(where $d$ is data size in bytes) | 1 + length of base addressing mode   |
| Autoincrement                        | (rn)+              | (r3)+    | Memory[r3]; r3 = r3 + d                                  | 1                                    |
| Autodecrement                        | -(rn)              | -(r3)    | r3 = r3 - d; Memory[r3]                                  | 1                                    |
| Autoincrement deferred               | @(rn)+             | @(r3)+   | Memory[Memory[r3]]; r3 = r3 + d                          | 1                                    |

**Figure K.50 Definition and length of the VAX operand specifiers.** The length of each addressing mode is 1 byte plus the length of any displacement or immediate field needed by the mode. Literal mode uses a special 2-bit tag and the remaining 6 bits encode the constant value. If the constant is too big, it must use the immediate addressing mode. Note that the length of an immediate operand is dictated by the length of the data type indicated in the opcode, not the value of the immediate. The symbol  $d$  in the last four modes represents the length of the data in bytes;  $d$  is 4 for 32-bit add.

---

**Example** How long is the following instruction?

addl3 r1,737(r2),(r3)[r4]

The name addl3 means a 32-bit add instruction with three operands. Assume the length of the VAX opcode is 1 byte.

**Answer** The first operand specifier—r1—indicates register addressing and is 1 byte long. The second operand specifier—737(r2)—indicates displacement addressing and has two parts: The first part is a byte that specifies the word displacement addressing mode and base register (r2); the second part is the 2-byte-long displacement (737). The third operand specifier—(r3)[r4]—also has two parts: The first byte specifies register deferred addressing mode ((r3)), and the second byte specifies the Index register and the use of indexed addressing ([r4]). Thus, the total length of the instruction is  $1 + (1) + (1 + 2) + (1 + 1) = 7$  bytes.

---

In this example instruction, we show the VAX destination operand on the left and the source operands on the right, just as we show MIPS code. The VAX assembler actually expects operands in the opposite order, but we felt it would be less confusing to keep the destination on the left for both machines. Obviously, left or right orientation is arbitrary; the only requirement is consistency.

**Elaboration** Because the PC is 1 of the 16 registers that can be selected in a VAX addressing mode, 4 of the 22 VAX addressing modes are synthesized from other addressing modes. Using the PC as the chosen register in each case, immediate addressing is really autoincrement, PC-relative is displacement, absolute is autoincrement deferred, and relative deferred is displacement deferred.

## Encoding VAX Instructions

Given the independence of the operations and addressing modes, the encoding of instructions is quite different from MIPS.

VAX instructions begin with a single byte opcode containing the operation and the number of operands. The operands follow the opcode. Each operand begins with a single byte, called the *address specifier*, that describes the addressing mode for that operand. For a simple addressing mode, such as register addressing, this byte specifies the register number as well as the mode (see the rightmost column in Figure K.50). In other cases, this initial byte can be followed by many more bytes to specify the rest of the address information.

As a specific example, let's show the encoding of the add instruction from the example on page K-24:

addl3 r1,737(r2),(r3)[r4]

Assume that this instruction starts at location 201.

Figure K.51 shows the encoding. Note that the operands are stored in memory in opposite order to the assembly code above. The execution of VAX instructions

| Byte address | Contents at each byte                             | Machine code      |
|--------------|---------------------------------------------------|-------------------|
| 201          | Opcode containing addl3                           | c1 <sub>hex</sub> |
| 202          | Index mode specifier for [r4]                     | 44 <sub>hex</sub> |
| 203          | Register indirect mode specifier for (r3)         | 63 <sub>hex</sub> |
| 204          | Word displacement mode specifier using r2 as base | c2 <sub>hex</sub> |
| 205          | The 16-bit constant 737                           | e1 <sub>hex</sub> |
| 206          |                                                   | 02 <sub>hex</sub> |
| 207          | Register mode specifier for r1                    | 51 <sub>hex</sub> |

**Figure K.51** The encoding of the VAX instruction addl3 r1,737(r2),(r3)[r4], assuming it starts at address 201. To satisfy your curiosity, the right column shows the actual VAX encoding in hexadecimal notation. Note that the 16-bit constant 737<sub>ten</sub> takes 2 bytes.

begins with fetching the source operands, so it makes sense for them to come first. Order is not important in fixed-length instructions like MIPS, since the source and destination operands are easily found within a 32-bit word.

The first byte, at location 201, is the opcode. The next byte, at location 202, is a specifier for the index mode using register r4. Like many of the other specifiers, the left 4 bits of the specifier give the mode and the right 4 bits give the register used in that mode. Since addl3 is a 4-byte operation, r4 will be multiplied by 4 and added to whatever address is specified next. In this case it is register deferred addressing using register r3. Thus, bytes 202 and 203 combined define the third operand in the assembly code.

The following byte, at address 204, is a specifier for word displacement addressing using register r2 as the base register. This specifier tells the VAX that the following two bytes, locations 205 and 206, contain a 16-bit address to be added to r2.

The final byte of the instruction gives the destination operand, and this specifier selects register addressing using register r1.

Such variability in addressing means that a single VAX operation can have many different lengths; for example, an integer add varies from 3 bytes to 19 bytes. VAX implementations must decode the first operand before they can find the second, and so implementors are strongly tempted to take 1 clock cycle to decode each operand; thus, this sophisticated instruction set architecture can result in higher clock cycles per instruction, even when using simple addresses.

## VAX Operations

In keeping with its philosophy, the VAX has a large number of operations as well as a large number of addressing modes. We review a few here to give the flavor of the machine.

Given the power of the addressing modes, the VAX *move* instruction performs several operations found in other machines. It transfers data between any two addressable locations and subsumes load, store, register-register moves, and

memory-memory moves as special cases. The first letter of the VAX data type (b, w, l, f, q, d, g, c in Figure K.49) is appended to the acronym mov to determine the size of the data. One special move, called *move address*, moves the 32-bit *address* of the operand rather than the data. It uses the acronym mova.

The arithmetic operations of MIPS are also found in the VAX, with two major differences. First, the type of the data is attached to the name. Thus, addb, addw, and addl operate on 8-bit, 16-bit, and 32-bit data in memory or registers, respectively; MIPS has a single add instruction that operates only on the full 32-bit register. The second difference is that to reduce code size the add instruction specifies the number of unique operands; MIPS always specifies three even if one operand is redundant. For example, the MIPS instruction

```
add $1, $1, $2
```

takes 32 bits like all MIPS instructions, but the VAX instruction

```
addl2 r1, r2
```

uses r1 for both the destination and a source, taking just 24 bits: 8 bits for the opcode and 8 bits each for the two register specifiers.

#### *Number of Operations*

Now we can show how VAX instruction names are formed:

$$(\text{operation})(\text{datatype}) \left( \frac{2}{3} \right)$$

The operation add works with data types byte, word, long, float, and double and comes in versions for either 2 or 3 unique operands, so the following instructions are all found in the VAX:

|       |       |       |       |
|-------|-------|-------|-------|
| addb2 | addw2 | addl2 | addf2 |
| addb3 | addw3 | addl3 | addf3 |

Accounting for all addressing modes (but ignoring register numbers and immediate values) and limiting to just byte, word, and long, there are more than 30,000 versions of integer add in the VAX; MIPS has just 4!

Another reason for the large number of VAX instructions is the instructions that either replace sequences of instructions or take fewer bytes to represent a single instruction. Here are four such examples (\* means the data type):

| VAX operation | Example  | Meaning                   |
|---------------|----------|---------------------------|
| clr*          | clr1 r3  | r3 = 0                    |
| inc*          | incl r3  | r3 = r3+1                 |
| dec*          | decl r3  | r3 = r3-1                 |
| push*         | pushl r3 | sp = sp-4; Memory[sp]=r3; |

The *push* instruction in the last row is exactly the same as using the move instruction with autodecrement addressing on the stack pointer:

```
movl - (sp), r3
```

Brevity is the advantage of *pushl*: It is 1 byte shorter since *sp* is implied.

### *Branches, Jumps, and Procedure Calls*

The VAX branch instructions are related to the arithmetic instructions because the branch instructions rely on *condition codes*. Condition codes are set as a side effect of an operation, and they indicate whether the result is positive, negative, or zero or if an overflow occurred. Most instructions set the VAX condition codes according to their result; instructions without results, such as branches, do not. The VAX condition codes are N (Negative), Z (Zero), V (oVerflow), and C (Carry). There is also a *compare* instruction *cmp\** just to set the condition codes for a subsequent branch.

The VAX branch instructions include all conditions. Popular branch instructions include *beql(=)*, *bneq(≠)*, *blss(<)*, *bleq(≤)*, *bgtr(>)*, and *bgeq(≥)*, which do just what you would expect. There are also unconditional branches whose name is determined by the size of the PC-relative offset. Thus, *brb* (*branch byte*) has an 8-bit displacement, and *brw* (*branch word*) has a 16-bit displacement.

The final major category we cover here is the procedure *call and return* instructions. Unlike the MIPS architecture, these elaborate instructions can take dozens of clock cycles to execute. The next two sections show how they work, but we need to explain the purpose of the pointers associated with the stack manipulated by *call*s and *ret*. The *stack pointer*, *sp*, is just like the stack pointer in MIPS; it points to the top of the stack. The *argument pointer*, *ap*, points to the base of the list of arguments or parameters in memory that are passed to the procedure. The *frame pointer*, *fp*, points to the base of the local variables of the procedure that are kept in memory (the *stack frame*). The VAX call and return instructions manipulate these pointers to maintain the stack in proper condition across procedure calls and to provide convenient base registers to use when accessing memory operands. As we shall see, call and return also save and restore the general-purpose registers as well as the program counter. Figure K.52 gives a further sampling of the VAX instruction set.

## An Example to Put It All Together: *swap*

To see programming in VAX assembly language, we translate two C procedures, *swap* and *sort*. The C code for *swap* is reproduced in Figure K.53. The next section covers *sort*.

We describe the *swap* procedure in three general steps of assembly language programming:

1. Allocate registers to program variables.
2. Produce code for the body of the procedure.
3. Preserve registers across the procedure invocation.

| Instruction type   | Example                                                                                       | Instruction meaning                                                                |
|--------------------|-----------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------|
| Data transfers     | Move data between byte, half-word, word, or double-word operands; * is data type              |                                                                                    |
|                    | mov*                                                                                          | Move between two operands                                                          |
|                    | movzb*                                                                                        | Move a byte to a half word or word, extending it with zeros                        |
|                    | mov a*                                                                                        | Move the 32-bit address of an operand; data type is last                           |
|                    | push*                                                                                         | Push operand onto stack                                                            |
| Arithmetic/logical | Operations on integer or logical bytes, half words (16 bits), words (32 bits); * is data type |                                                                                    |
|                    | add*_                                                                                         | Add with 2 or 3 operands                                                           |
|                    | cmp*                                                                                          | Compare and set condition codes                                                    |
|                    | tst*                                                                                          | Compare to zero and set condition codes                                            |
|                    | ash*                                                                                          | Arithmetic shift                                                                   |
|                    | clr*                                                                                          | Clear                                                                              |
|                    | cvtb*                                                                                         | Sign-extend byte to size of data type                                              |
| Control            | Conditional and unconditional branches                                                        |                                                                                    |
|                    | beql, bneq                                                                                    | Branch equal, branch not equal                                                     |
|                    | breq, bgeq                                                                                    | Branch less than or equal, branch greater than or equal                            |
|                    | brb, brw                                                                                      | Unconditional branch with an 8-bit or 16-bit address                               |
|                    | jmp                                                                                           | Jump using any addressing mode to specify target                                   |
|                    | aobreq                                                                                        | Add one to operand; branch if result $\leq$ second operand                         |
|                    | case_                                                                                         | Jump based on case selector                                                        |
| Procedure          | Call/return from procedure                                                                    |                                                                                    |
|                    | calls                                                                                         | Call procedure with arguments on stack (see “A Longer Example: sort” on page K-33) |
|                    | callg                                                                                         | Call procedure with FORTRAN-style parameter list                                   |
|                    | jsb                                                                                           | Jump to subroutine, saving return address (like MIPS jal)                          |
|                    | ret                                                                                           | Return from procedure call                                                         |
| Floating point     | Floating-point operations on D, F, G, and H formats                                           |                                                                                    |
|                    | addir_                                                                                        | Add double-precision D-format floating numbers                                     |
|                    | subd_                                                                                         | Subtract double-precision D-format floating numbers                                |
|                    | mult_                                                                                         | Multiply single-precision F-format floating point                                  |
|                    | polyf                                                                                         | Evaluate a polynomial using table of coefficients in F format                      |
| Other              | Special operations                                                                            |                                                                                    |
|                    | crc                                                                                           | Calculate cyclic redundancy check                                                  |
|                    | insque                                                                                        | Insert a queue entry into a queue                                                  |

**Figure K.52 Classes of VAX instructions with examples.** The asterisk stands for multiple data types: b, w, l, d, f, g, h, and q. The underline, as in addd\_, means there are 2-operand (addir2) and 3-operand (addir3) forms of this instruction.

---

```
swap(int v[], int k)
{
 int temp;
 temp = v[k];
 v[k] = v[k + 1];
 v[k + 1] = temp;
}
```

---

**Figure K.53 A C procedure that swaps two locations in memory.** This procedure will be used in the sorting example in the next section.

The VAX code for these procedures is based on code produced by the VMS C compiler using optimization.

#### *Register Allocation for swap*

In contrast to MIPS, VAX parameters are normally allocated to memory, so this step of assembly language programming is more properly called “variable allocation.” The standard VAX convention on parameter passing is to use the stack. The two parameters,  $v[]$  and  $k$ , can be accessed using register  $ap$ , the argument pointer: The address  $4(ap)$  corresponds to  $v[]$  and  $8(ap)$  corresponds to  $k$ . Remember that with byte addressing the address of sequential 4-byte words differs by 4. The only other variable is  $temp$ , which we associate with register  $r3$ .

#### *Code for the Body of the Procedure swap*

The remaining lines of C code in *swap* are

```
temp = v[k];
v[k] = v[k + 1];
v[k + 1] = temp;
```

Since this program uses  $v[]$  and  $k$  several times, to make the programs run faster the VAX compiler first moves both parameters into registers:

```
movl r2, 4(ap) ;r2 = v[]
movl r1, 8(ap) ;r1 = k
```

Note that we follow the VAX convention of using a semicolon to start a comment; the MIPS comment symbol # represents a constant operand in VAX assembly language.

The VAX has indexed addressing, so we can use index k without converting it to a byte address. The VAX code is then straightforward:

```
movl r3, (r2)[r1] ; r3 (temp) = v[k]
addl3 r0, #1,8(ap) ; r0 = k + 1
movl (r2)[r1],(r2)[r0] ; v[k] = v[r0] (v[k + 1])
movl (r2)[r0],r3 ; v[k + 1] = r3 (temp)
```

Unlike the MIPS code, which is basically two loads and two stores, the key VAX code is one memory-to-register move, one memory-to-memory move, and one register-to-memory move. Note that the `addl3` instruction shows the flexibility of the VAX addressing modes: It adds the constant 1 to a memory operand and places the result in a register.

Now we have allocated storage and written the code to perform the operations of the procedure. The only missing item is the code that preserves registers across the routine that calls `swap`.

#### *Preserving Registers across Procedure Invocation of swap*

The VAX has a pair of instructions that preserve registers, `calls` and `ret`. This example shows how they work.

The VAX C compiler uses a form of callee convention. Examining the code above, we see that the values in registers `r0`, `r1`, `r2`, and `r3` must be saved so that they can later be restored. The `calls` instruction expects a 16-bit mask at the beginning of the procedure to determine which registers are saved: if bit  $i$  is set in the mask, then register  $i$  is saved on the stack by the `calls` instruction. In addition, `calls` saves this mask on the stack to allow the return instruction (`ret`) to restore the proper registers. Thus, the `calls` executed by the caller does the saving, but the callee sets the call mask to indicate what should be saved.

One of the operands for `calls` gives the number of parameters being passed, so that `calls` can adjust the pointers associated with the stack: the argument pointer (`ap`), frame pointer (`fp`), and stack pointer (`sp`). Of course, `calls` also saves the program counter so that the procedure can return!

Thus, to preserve these four registers for `swap`, we just add the mask at the beginning of the procedure, letting the `calls` instruction in the caller do all the work:

```
.word ^m<r0,r1,r2,r3> ;set bits in mask for 0,1,2,3
```

This directive tells the assembler to place a 16-bit constant with the proper bits set to save registers `r0` through `r3`.

The return instruction undoes the work of `calls`. When finished, `ret` sets the stack pointer from the current frame pointer to pop everything `calls` placed on the stack. Along the way, it restores the register values saved by `calls`, including those marked by the mask and old values of the `fp`, `ap`, and `pc`.

To complete the procedure swap, we just add one instruction:

```
ret ; restore registers and return
```

### *The Full Procedure swap*

We are now ready for the whole routine. Figure K.54 identifies each block of code with its purpose in the procedure, with the MIPS code on the left and the VAX code on the right. This example shows the advantage of the scaled indexed addressing and the sophisticated call and return instructions of the VAX in reducing the number of lines of code. The 17 lines of MIPS assembly code became 8 lines of VAX assembly code. It also shows that passing parameters in memory results in extra memory accesses.

Keep in mind that the number of instructions executed is not the same as performance; the fallacy on page K-38 makes this point.

Note that VAX software follows a convention of treating registers r0 and r1 as temporaries that are not saved across a procedure call, so the VMS C compiler does include registers r0 and r1 in the register saving mask. Also, the C compiler should have used r1 instead of 8(ap) in the addl3 instruction; such examples inspire computer architects to try to write compilers!

| <b>MIPS versus VAX</b>    |  |                             |  |
|---------------------------|--|-----------------------------|--|
| Saving register           |  |                             |  |
| swap: addi \$29,\$29, -12 |  | swap: .word ^m<r0,r1,r2,r3> |  |
| sw \$2, 0(\$29)           |  |                             |  |
| sw \$15, 4(\$29)          |  |                             |  |
| sw \$16, 8(\$29)          |  |                             |  |
| Procedure body            |  |                             |  |
| muli \$2, \$5,4           |  | movl r2, 4(a)               |  |
| add \$2, \$4,\$2          |  | movl r1, 8(a)               |  |
| lw \$15, 0(\$2)           |  | movl r3, (r2)[r1]           |  |
| lw \$16, 4(\$2)           |  | addl3 r0, #1,8(ap)          |  |
| sw \$16, 0(\$2)           |  | movl (r2)[r1],(r2)[r0]      |  |
| sw \$15, 4(\$2)           |  | movl (r2)[r0],r3            |  |
| Restoring registers       |  |                             |  |
| lw \$2, 0(\$29)           |  |                             |  |
| lw \$15, 4(\$29)          |  |                             |  |
| lw \$16, 8(\$29)          |  |                             |  |
| addi \$29,\$29, 12        |  |                             |  |
| Procedure return          |  |                             |  |
| jr \$31                   |  | ret                         |  |

**Figure K.54** MIPS versus VAX assembly code of the procedure swap in Figure K.53 on page K-30.

## A Longer Example: sort

We show the longer example of the sort procedure. Figure K.55 shows the C version of the program. Once again we present this procedure in several steps, concluding with a side-by-side comparison to MIPS code.

### Register Allocation for sort

The two parameters of the procedure sort, v and n, are found in the stack in locations 4(ap) and 8(ap), respectively. The two local variables are assigned to registers: i to r6 and j to r4. Because the two parameters are referenced frequently in the code, the VMS C compiler copies the *address* of these parameters into registers upon entering the procedure:

```
movl r7,8(ap) ;move address of n into r7
movl r5,4(ap) ;move address of v into r5
```

It would seem that moving the *value* of the operand to a register would be more useful than its address, but once again we bow to the decision of the VMS C compiler. Apparently the compiler cannot be sure that v and n don't overlap in memory.

### Code for the Body of the sort Procedure

The procedure body consists of two nested *for* loops and a call to swap, which includes parameters. Let's unwrap the code from the outside to the middle.

#### The Outer Loop

The first translation step is the first for loop:

```
for (i = 0; i < n; i = i + 1) {
```

Recall that the C for statement has three parts: initialization, loop test, and iteration increment. It takes just one instruction to initialize i to 0, the first part of the for statement:

```
clr1 r6 ;i = 0
```

---

```
sort (int v[], int n)
{
 int i, j;
 for (i = 0; i < n; i = i + 1) {
 for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j = j - 1)
 { swap(v,j);
 }
 }
}
```

---

**Figure K.55** A C procedure that performs a bubble sort on the array v.

It also takes just one instruction to increment  $i$ , the last part of the for:

```
incl r6 ;i = i + 1
```

The loop should be exited if  $i < n$  is *false*, or said another way, exit the loop if  $i \geq n$ . This test takes two instructions:

```
for1tst: cmpl r6,(r7) ;compare r6 and memory[r7] (i:n)
 bgeq exit1 ;go to exit1 if r6 \geq mem[r7] (i \geq n)
```

Note that `cmpl` sets the condition codes for use by the conditional branch instruction `bgeq`.

The bottom of the loop just jumps back to the loop test:

```
brb for1tst ;branch to test of outer loop
exit1:
```

The skeleton code of the first for loop is then

```
clr1 r6 ;i = 0
for1tst: cmpl r6,(r7) ;compare r6 and memory[r7] (i:n)
 bgeq exit1 ;go to exit1 if r6 \geq mem[r7] (i \geq n)
 ...
 (body of first for loop)
 ...
incl r6 ;i = i + 1
brb for1tst ;branch to test of outer loop
exit1:
```

### The Inner Loop

The second for loop is

```
for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j = j - 1) {
```

The initialization portion of this loop is again one instruction:

```
subl3 r4,r6,#1 ;j = i - 1
```

The decrement of  $j$  is also one instruction:

```
decl r4 ;j = j - 1
```

The loop test has two parts. We exit the loop if either condition fails, so the first test must exit the loop if it fails ( $j < 0$ ):

```
for2tst: blss exit2 ;go to exit2 if r4 < 0 (j < 0)
```

Notice that there is no explicit comparison. The lack of comparison is a benefit of condition codes, with the conditions being set as a side effect of the prior instruction. This branch skips over the second condition test.

The second test exits if  $v[j] > v[j + 1]$  is false, or exits if  $v[j] \leq v[j + 1]$ . First we load  $v$  and put  $j + 1$  into registers:

```
movl r3,(r5) ;r3 = Memory[r5] (r3 = v)
addl3 r2,r4,#1 ;r2 = r4 + 1 (r2 = j + 1)
```

Register indirect addressing is used to get the operand pointed to by r5.

Once again the index addressing mode means we can use indices without converting to the byte address, so the two instructions for  $v[j] \leq v[j+1]$  are

```
cmp1 (r3)[r4],(r3)[r2] ;v[r4]:v[r2](v[j]:v[j+1])
bleq exit2 ;go to exit2 if v[j] \leq v[j+1]
```

The bottom of the loop jumps back to the full loop test:

```
brb for2tst# jump to test of inner loop
```

Combining the pieces, the second for loop looks like this:

```
subl3 r4,r6,#1 ;j = i - 1
for2tst: blss exit2 ;go to exit2 if r4 < 0 (j < 0)
 movl r3,(r5) ;r3 = Memory[r5] (r3 = v)
 addl3 r2,r4,#1 ;r2 = r4 + 1 (r2 = j + 1)
 cmp1 (r3)[r4],(r3)[r2];v[r4]:v[r2]
 bleq exit2 ;go to exit2 if v[j] \leq v[j+1]
 ...
 (body of second for loop) ...
 dec1 r4 ;j = j - 1
 brb for2tst ;jump to test of inner loop
exit2:
```

Notice that the instruction `blss` (at the top of the loop) is testing the condition codes based on the new value of `r4` (`j`), set either by the `subl3` before entering the loop or by the `dec1` at the bottom of the loop.

### The Procedure Call

The next step is the body of the second for loop:

```
swap(v,j);
```

Calling swap is easy enough:

```
calls #2,swap
```

The constant 2 indicates the number of parameters pushed on the stack.

### Passing Parameters

The C compiler passes variables on the stack, so we pass the parameters to `swap` with these two instructions:

```
pushl (r5) ;first swap parameter is v
pushl r4 ;second swap parameter is j
```

Register indirect addressing is used to get the operand of the first instruction.

### *Preserving Registers across Procedure Invocation of sort*

The only remaining code is the saving and restoring of registers using the callee save convention. This procedure uses registers `r2` through `r7`, so we add a mask with those bits set:

```
.word m<r2,r3,r4,r5,r6,r7>; set mask for registers 2-7
```

Since ret will undo all the operations, we just tack it on the end of the procedure.

### *The Full Procedure sort*

Now we put all the pieces together in Figure K.56. To make the code easier to follow, once again we identify each block of code with its purpose in the procedure and list the MIPS and VAX code side by side. In this example, 11 lines of the sort procedure in C become the 44 lines in the MIPS assembly language and 20 lines in VAX assembly language. The biggest VAX advantages are in register saving and restoring and indexed addressing.

## Fallacies and Pitfalls

*The ability to simplify means to eliminate the unnecessary so that the necessary may speak.*

**Hans Hoffman**  
*Search for the Real (1967)*

**Fallacy** *It is possible to design a flawless architecture.*

All architecture design involves trade-offs made in the context of a set of hardware and software technologies. Over time those technologies are likely to change, and decisions that may have been correct at one time later look like mistakes. For example, in 1975 the VAX designers overemphasized the importance of code size efficiency and underestimated how important ease of decoding and pipelining would be 10 years later. And, almost all architectures eventually succumb to the lack of sufficient address space. Avoiding these problems in the long run, however, would probably mean compromising the efficiency of the architecture in the short run.

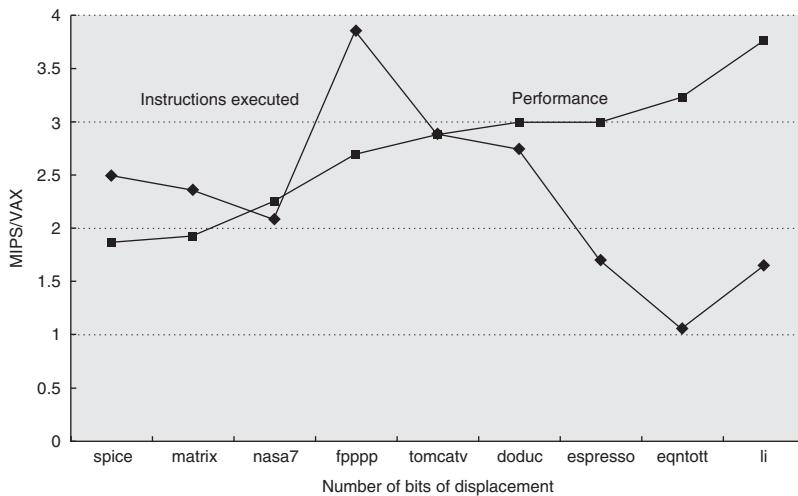
**Fallacy** *An architecture with flaws cannot be successful.*

The IBM 360 is often criticized in the literature—the branches are not PC-relative, and the address is too small in displacement addressing. Yet, the machine has been an enormous success because it correctly handled several new problems. First, the architecture has a large amount of address space. Second, it is byte addressed and handles bytes well. Third, it is a general-purpose register machine. Finally, it is simple enough to be efficiently implemented across a wide performance and cost range.

The Intel 8086 provides an even more dramatic example. The 8086 architecture is the only widespread architecture in existence today that is not truly a general-purpose register machine. Furthermore, the segmented address space of the 8086 causes major problems for both programmers and compiler writers. Nevertheless, the 8086 architecture—because of its selection as the microprocessor in the IBM PC—has been enormously successful.

| MIPS versus VAX                                                                                                                                                                                               |  |  |                                                       |  |  |  |  |  |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|--|-------------------------------------------------------|--|--|--|--|--|
| Saving registers                                                                                                                                                                                              |  |  |                                                       |  |  |  |  |  |
| sort:                                                                                                                                                                                                         |  |  | sort: .word ^m<r2,r3,r4,r5,r6,r7>                     |  |  |  |  |  |
| addi \$29,\$29, -36<br>sw \$15, 0(\$29)<br>sw \$16, 4(\$29)<br>sw \$17, 8(\$29)<br>sw \$18,12(\$29)<br>sw \$19,16(\$29)<br>sw \$20,20(\$29)<br>sw \$24,24(\$29)<br>sw \$25,28(\$29)<br>sw \$31,32(\$29)       |  |  |                                                       |  |  |  |  |  |
| Procedure body                                                                                                                                                                                                |  |  |                                                       |  |  |  |  |  |
| Move parameters                                                                                                                                                                                               |  |  |                                                       |  |  |  |  |  |
| move \$18, \$4<br>move \$20, \$5                                                                                                                                                                              |  |  | moval r7,8(ap)<br>moval r5,4(ap)                      |  |  |  |  |  |
| Outer loop                                                                                                                                                                                                    |  |  |                                                       |  |  |  |  |  |
| for1tst: add \$19, \$0, \$0<br>slt \$8, \$19, \$20<br>beq \$8, \$0, exit1                                                                                                                                     |  |  | for1tst: clrl r6<br>cmp1 r6,(r7)<br>bgeq exit1        |  |  |  |  |  |
| Inner loop                                                                                                                                                                                                    |  |  |                                                       |  |  |  |  |  |
| for2tst: addi \$17, \$19, -1<br>slti \$8, \$17, 0<br>bne \$8, \$0, exit2<br>multi \$15, \$17, 4<br>add \$16, \$18, \$15<br>lw \$24, 0(\$16)<br>lw \$25, 4(\$16)<br>slt \$8, \$25, \$24<br>beq \$8, \$0, exit2 |  |  | for2tst: subl3 r4,r6,#1<br>blss exit2<br>movl r3,(r5) |  |  |  |  |  |
| Pass parameters and call                                                                                                                                                                                      |  |  |                                                       |  |  |  |  |  |
| move \$4, \$18<br>move \$5, \$17<br>jal swap                                                                                                                                                                  |  |  | pushl (r5)<br>pushl r4<br>calls #2,swap               |  |  |  |  |  |
| Inner loop                                                                                                                                                                                                    |  |  |                                                       |  |  |  |  |  |
| addi \$17, \$17, -1<br>j for2tst                                                                                                                                                                              |  |  | decl r4<br>brb for2tst                                |  |  |  |  |  |
| Outer loop                                                                                                                                                                                                    |  |  |                                                       |  |  |  |  |  |
| exit2: addi \$19, \$19, 1<br>j for1tst                                                                                                                                                                        |  |  | exit2: incl r6<br>brb for1tst                         |  |  |  |  |  |
| Restoring registers                                                                                                                                                                                           |  |  |                                                       |  |  |  |  |  |
| exit1: lw \$15,0(\$29)<br>lw \$16, 4(\$29)<br>lw \$17, 8(\$29)<br>lw \$18,12(\$29)<br>lw \$19,16(\$29)<br>lw \$20,20(\$29)<br>lw \$24,24(\$29)<br>lw \$25,28(\$29)<br>lw \$31,32(\$29)<br>addi \$29,\$29, 36  |  |  |                                                       |  |  |  |  |  |
| Procedure return                                                                                                                                                                                              |  |  |                                                       |  |  |  |  |  |
| jr \$31                                                                                                                                                                                                       |  |  | exit1: ret                                            |  |  |  |  |  |

Figure K.56 MIPS32 versus VAX assembly version of procedure sort in Figure K.55 on page K-33.



**Figure K.57** Ratio of MIPS M2000 to VAX 8700 in instructions executed and performance in clock cycles using SPEC89 programs. On average, MIPS executes a little over twice as many instructions as the VAX, but the CPI for the VAX is almost six times the MIPS CPI, yielding almost a threefold performance advantage. (Based on data from “Performance from Architecture: Comparing a RISC and CISC with Similar Hardware Organization,” by D. Bhandarkar and D. Clark, in *Proc. Symp. Architectural Support for Programming Languages and Operating Systems IV*, 1991.)

**Fallacy** *The architecture that executes fewer instructions is faster.*

Designers of VAX machines performed a quantitative comparison of VAX and MIPS for implementations with comparable organizations, the VAX 8700 and the MIPS M2000. Figure K.57 shows the ratio of the number of instructions executed and the ratio of performance measured in clock cycles. MIPS executes about twice as many instructions as the VAX while the MIPS M2000 has almost three times the performance of the VAX 8700.

### Concluding Remarks

*The Virtual Address eXtension of the PDP-11 architecture ... provides a virtual address of about 4.3 gigabytes which, even given the rapid improvement of memory technology, should be adequate far into the future.*

**William Strecker**

*“VAX-11/780—A Virtual Address Extension to the PDP-11 Family,”  
AFIPS Proc., National Computer Conference (1978)*

We have seen that instruction sets can vary quite dramatically, both in how they access operands and in the operations that can be performed by a single instruction. Figure K.58 compares instruction usage for both architectures for two programs; even very different architectures behave similarly in their use of instruction classes.

| Program | Machine | Branch | Arithmetic/<br>logical | Data<br>transfer | Floating<br>point | Totals |
|---------|---------|--------|------------------------|------------------|-------------------|--------|
| gcc     | VAX     | 30%    | 40%                    | 19%              |                   | 89%    |
|         | MIPS    | 24%    | 35%                    | 27%              |                   | 86%    |
| spice   | VAX     | 18%    | 23%                    | 15%              | 23%               | 79%    |
|         | MIPS    | 4%     | 29%                    | 35%              | 15%               | 83%    |

**Figure K.58** The frequency of instruction distribution for two programs on VAX and MIPS.

A product of its time, the VAX emphasis on code density and complex operations and addressing modes conflicts with the current emphasis on easy decoding, simple operations and addressing modes, and pipelined performance.

With more than 600,000 sold, the VAX architecture has had a very successful run. In 1991, DEC made the transition from VAX to Alpha.

Orthogonality is key to the VAX architecture; the opcode is independent of the addressing modes, which are independent of the data types and even the number of unique operands. Thus, a few hundred operations expand to hundreds of thousands of instructions when accounting for the data types, operand counts, and addressing modes.

## Exercises

- K.1 [3] <K.4> The following VAX instruction decrements the location pointed to be register r5:

decl (r5)

What is the single MIPS instruction, or if it cannot be represented in a single instruction, the shortest sequence of MIPS instructions, that performs the same operation? What are the lengths of the instructions on each machine?

- K.2 [5] <K.4> This exercise is the same as Exercise K.1, except this VAX instruction clears a location using autoincrement deferred addressing:

clr1 @(r5)+

- K.3 [5] <K.4> This exercise is the same as Exercise K.1, except this VAX instruction adds 1 to register r5, placing the sum back in register r5, compares the sum to register r6, and then branches to L1 if  $r5 < r6$ :

aoblss r6, r5, L1 # r5 = r5 + 1; if (r5 < r6) goto L1.

- K.4 [5] <K.4> Show the single VAX instruction, or minimal sequence of instructions, for this C statement:

a = b + 100;

Assume a corresponds to register r3 and b corresponds to register r4.

- K.5 [10] <K.4> Show the single VAX instruction, or minimal sequence of instructions, for this C statement:

$x[i + 1] = x[i] + c;$

Assume c corresponds to register r3, i to register r4, and x is an array of 32-bit words beginning at memory location  $4,000,000_{\text{ten}}$ .

**K.5****The IBM 360/370 Architecture for Mainframe Computers****Introduction**

The term “computer architecture” was coined by IBM in 1964 for use with the IBM 360. Amdahl, Blaauw, and Brooks [1964] used the term to refer to the programmer-visible portion of the instruction set. They believed that a family of machines of the same architecture should be able to run the same software. Although this idea may seem obvious to us today, it was quite novel at the time. IBM, even though it was the leading company in the industry, had five different architectures before the 360. Thus, the notion of a company standardizing on a single architecture was a radical one. The 360 designers hoped that six different divisions of IBM could be brought together by defining a common architecture. Their definition of *architecture* was

... the structure of a computer that a machine language programmer must understand to write a correct (timing independent) program for that machine.

The term “machine language programmer” meant that compatibility would hold, even in assembly language, while “timing independent” allowed different implementations.

The IBM 360 was introduced in 1964 with six models and a 25:1 performance ratio. Amdahl, Blaauw, and Brooks [1964] discussed the architecture of the IBM 360 and the concept of permitting multiple object-code-compatible implementations. The notion of an instruction set architecture as we understand it today was the most important aspect of the 360. The architecture also introduced several important innovations, now in wide use:

1. 32-bit architecture
2. Byte-addressable memory with 8-bit bytes
3. 8-, 16-, 32-, and 64-bit data sizes
4. 32-bit single-precision and 64-bit double-precision floating-point data

In 1971, IBM shipped the first System/370 (models 155 and 165), which included a number of significant extensions of the 360, as discussed by Case and Padegs [1978], who also discussed the early history of System/360. The most important addition was virtual memory, though virtual memory 370 s did not ship until 1972, when a virtual memory operating system was ready. By 1978, the high-end 370 was several hundred times faster than the low-end 360 s shipped 10 years earlier. In 1984, the 24-bit addressing model built into the IBM 360 needed to be abandoned, and the 370-XA (eXtended Architecture) was introduced. While old 24-bit programs could be supported without change, several instructions could not function in the same manner when extended to a 32-bit addressing model (31-bit addresses supported) because they would not produce 31-bit addresses. Converting the operating system, which was written mostly in assembly language, was no doubt the biggest task.

Several studies of the IBM 360 and instruction measurement have been made. Shustek’s thesis [1978] is the best known and most complete study of the 360/370 architecture. He made several observations about instruction set complexity that

were not fully appreciated until some years later. Another important study of the 360 is the Toronto study by Alexander and Wortman [1975] done on an IBM 360 using 19 XPL programs.

### **System/360 Instruction Set**

The 360 instruction set is shown in the following tables, organized by instruction type and format. System/370 contains 15 additional user instructions.

#### *Integer/Logical and Floating-Point R-R Instructions*

The \* indicates the instruction is floating point, and may be either D (double precision) or E (single precision).

| <b>Instruction</b> | <b>Description</b>        |
|--------------------|---------------------------|
| ALR                | Add logical register      |
| AR                 | Add register              |
| A*R                | FP addition               |
| CLR                | Compare logical register  |
| CR                 | Compare register          |
| C*R                | FP compare                |
| DR                 | Divide register           |
| D*R                | FP divide                 |
| H*R                | FP halve                  |
| LCR                | Load complement register  |
| LC*R               | Load complement           |
| LNR                | Load negative register    |
| LN*R               | Load negative             |
| LPR                | Load positive register    |
| LP*R               | Load positive             |
| LR                 | Load register             |
| L*R                | Load FP register          |
| LTR                | Load and test register    |
| LT*R               | Load and test FP register |
| MR                 | Multiply register         |
| M*R                | FP multiply               |
| NR                 | And register              |
| OR                 | Or register               |
| SLR                | Subtract logical register |
| SR                 | Subtract register         |
| S*R                | FP subtraction            |
| XR                 | Exclusive or register     |

*Branches and Status Setting R-R Instructions*

These are R-R format instructions that either branch or set some system status; several of them are privileged and legal only in supervisor mode.

| Instruction | Description      |
|-------------|------------------|
| BALR        | Branch and link  |
| BCTR        | Branch on count  |
| BCR         | Branch/condition |
| ISK         | Insert key       |
| SPM         | Set program mask |
| SSK         | Set storage key  |
| SVC         | Supervisor call  |

*Branches/Logical and Floating-Point Instructions—RX Format*

These are all RX format instructions. The symbol “+” means either a word operation (and then stands for nothing) or H (meaning half word); for example, A+ stands for the two opcodes A and AH. The “\*” represents D or E, standing for double- or single-precision floating point.

| Instruction | Description       |
|-------------|-------------------|
| A+          | Add               |
| A*          | FP add            |
| AL          | Add logical       |
| C+          | Compare           |
| C*          | FP compare        |
| CL          | Compare logical   |
| D           | Divide            |
| D*          | FP divide         |
| L+          | Load              |
| L*          | Load FP register  |
| M+          | Multiply          |
| M*          | FP multiply       |
| N           | And               |
| O           | Or                |
| S+          | Subtract          |
| S*          | FP subtract       |
| SL          | Subtract logical  |
| ST+         | Store             |
| ST*         | Store FP register |
| X           | Exclusive or      |

*Branches and Special Loads and Stores—RX Format*

| <b>Instruction</b> | <b>Description</b> |
|--------------------|--------------------|
| BAL                | Branch and link    |
| BC                 | Branch condition   |
| BCT                | Branch on count    |
| CVB                | Convert-binary     |
| CVD                | Convert-decimal    |
| EX                 | Execute            |
| IC                 | Insert character   |
| LA                 | Load address       |
| STC                | Store character    |

*RS and SI Format Instructions*

These are the RS and SI format instructions. The symbol “\*” may be A (arithmetic) or L (logical).

| <b>Instruction</b> | <b>Description</b>        |
|--------------------|---------------------------|
| BXH                | Branch/high               |
| BXLE               | Branch/low-equal          |
| CLI                | Compare logical immediate |
| HIO                | Halt I/O                  |
| LPSW               | Load PSW                  |
| LM                 | Load multiple             |
| MVI                | Move immediate            |
| NI                 | And immediate             |
| OI                 | Or immediate              |
| RDD                | Read direct               |
| SIO                | Start I/O                 |
| SL*                | Shift left A/L            |
| SLD*               | Shift left double A/L     |
| SR*                | Shift right A/L           |
| SRD*               | Shift right double A/L    |
| SSM                | Set system mask           |
| STM                | Store multiple            |
| TCH                | Test channel              |
| TIO                | Test I/O                  |
| TM                 | Test under mask           |
| TS                 | Test-and-set              |
| WRD                | Write direct              |
| XI                 | Exclusive or immediate    |

*SS Format Instructions*

These are add decimal or string instructions.

| <b>Instruction</b> | <b>Description</b>         |
|--------------------|----------------------------|
| AP                 | Add packed                 |
| CLC                | Compare logical chars      |
| CP                 | Compare packed             |
| DP                 | Divide packed              |
| ED                 | Edit                       |
| EDMK               | Edit and mark              |
| MP                 | Multiply packed            |
| MVC                | Move character             |
| MVN                | Move numeric               |
| MVO                | Move with offset           |
| MVZ                | Move zone                  |
| NC                 | And characters             |
| OC                 | Or characters              |
| PACK               | Pack (Character → decimal) |
| SP                 | Subtract packed            |
| TR                 | Translate                  |
| TRT                | Translate and test         |
| UNPK               | Unpack                     |
| XC                 | Exclusive or characters    |
| ZAP                | Zero and add packed        |

**360 Detailed Measurements**

Figure K.59 shows the frequency of instruction usage for four IBM 360 programs.

| <b>Instruction</b>        | <b>PLIC</b> | <b>FORTGO</b> | <b>PLIGO</b> | <b>COBOLGO</b> | <b>Average</b> |
|---------------------------|-------------|---------------|--------------|----------------|----------------|
| <b>Control</b>            | <b>32%</b>  | <b>13%</b>    | <b>5%</b>    | <b>16%</b>     | <b>16%</b>     |
| BC, BCR                   | 28%         | 13%           | 5%           | 14%            | 15%            |
| BAL, BALR                 | 3%          |               |              | 2%             | 1%             |
| <b>Arithmetic/logical</b> | <b>29%</b>  | <b>35%</b>    | <b>29%</b>   | <b>9%</b>      | <b>26%</b>     |
| A, AR                     | 3%          | 17%           | 21%          |                | 10%            |
| SR                        | 3%          | 7%            |              |                | 3%             |
| SLL                       |             | 6%            | 3%           |                | 2%             |
| LA                        | 8%          | 1%            | 1%           |                | 2%             |
| CLI                       | 7%          |               |              |                | 2%             |
| NI                        |             |               |              | 7%             | 2%             |
| C                         | 5%          | 4%            | 4%           | 0%             | 3%             |
| TM                        | 3%          | 1%            |              | 3%             | 2%             |
| MH                        |             |               | 2%           |                | 1%             |
| <b>Data transfer</b>      | <b>17%</b>  | <b>40%</b>    | <b>56%</b>   | <b>20%</b>     | <b>33%</b>     |
| L, LR                     | 7%          | 23%           | 28%          | 19%            | 19%            |
| MVI                       | 2%          |               | 16%          | 1%             | 5%             |
| ST                        | 3%          |               | 7%           |                | 3%             |
| LD                        |             | 7%            | 2%           |                | 2%             |
| STD                       |             | 7%            | 2%           |                | 2%             |
| LPDR                      |             | 3%            |              |                | 1%             |
| LH                        | 3%          |               |              |                | 1%             |
| IC                        | 2%          |               |              |                | 1%             |
| LTR                       |             | 1%            |              |                | 0%             |
| <b>Floating point</b>     |             | <b>7%</b>     |              |                | <b>2%</b>      |
| AD                        |             | 3%            |              |                | 1%             |
| MDR                       |             | 3%            |              |                | 1%             |
| <b>Decimal, string</b>    | <b>4%</b>   |               | <b>40%</b>   | <b>11%</b>     |                |
| MVC                       | 4%          |               | 7%           |                | 3%             |
| AP                        |             |               | 11%          |                | 3%             |
| ZAP                       |             |               | 9%           |                | 2%             |
| CVD                       |             |               | 5%           |                | 1%             |
| MP                        |             |               | 3%           |                | 1%             |
| CLC                       |             |               | 3%           |                | 1%             |
| CP                        |             |               | 2%           |                | 1%             |
| ED                        |             |               | 1%           |                | 0%             |
| <b>Total</b>              | <b>82%</b>  | <b>95%</b>    | <b>90%</b>   | <b>85%</b>     | <b>88%</b>     |

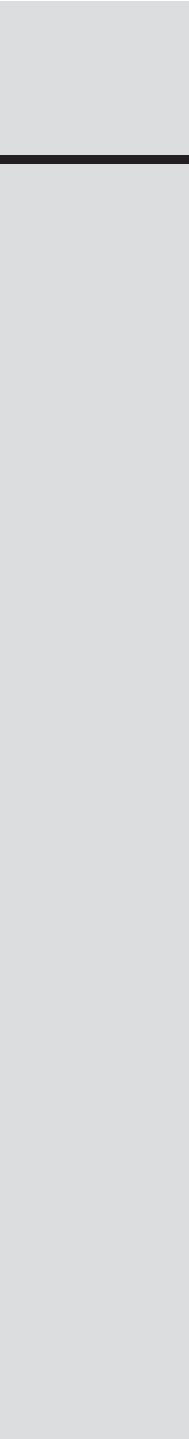
**Figure K.59** Distribution of instruction execution frequencies for the four 360 programs. All instructions with a frequency of execution greater than 1.5% are included. Immediate instructions, which operate on only a single byte, are included in the section that characterized their operation, rather than with the long character-string versions of the same operation. By comparison, the average frequencies for the major instruction classes of the VAX are 23% (control), 28% (arithmetic), 29% (data transfer), 7% (floating point), and 9% (decimal). Once again, a 1% entry in the average column can occur because of entries in the constituent columns. These programs are a compiler for the programming language PL-I and runtime systems for the programming languages FORTRAN, PL/I, and Cobol.

**K.6****Historical Perspective and References**

Section L.4 (available online) features a discussion on the evolution of instruction sets and includes references for further reading and exploration of related topics.

**Acknowledgments**

We would like to thank the following people for comments on drafts of this survey: Professor Steven B. Furber, University of Manchester; Dr. Dileep Bhandarkar, Intel Corporation; Dr. Earl Killian, Silicon Graphics/MIPS; and Dr. Hiokazu Takata, Mitsubishi Electric Corporation.



# L

---

## Advanced Concepts on Address Translation

by **Abhishek Bhattacharjee**

Appendix L is available online at <https://www.elsevier.com/books/computer-architecture/hennessy/978-0-12-811905-1>