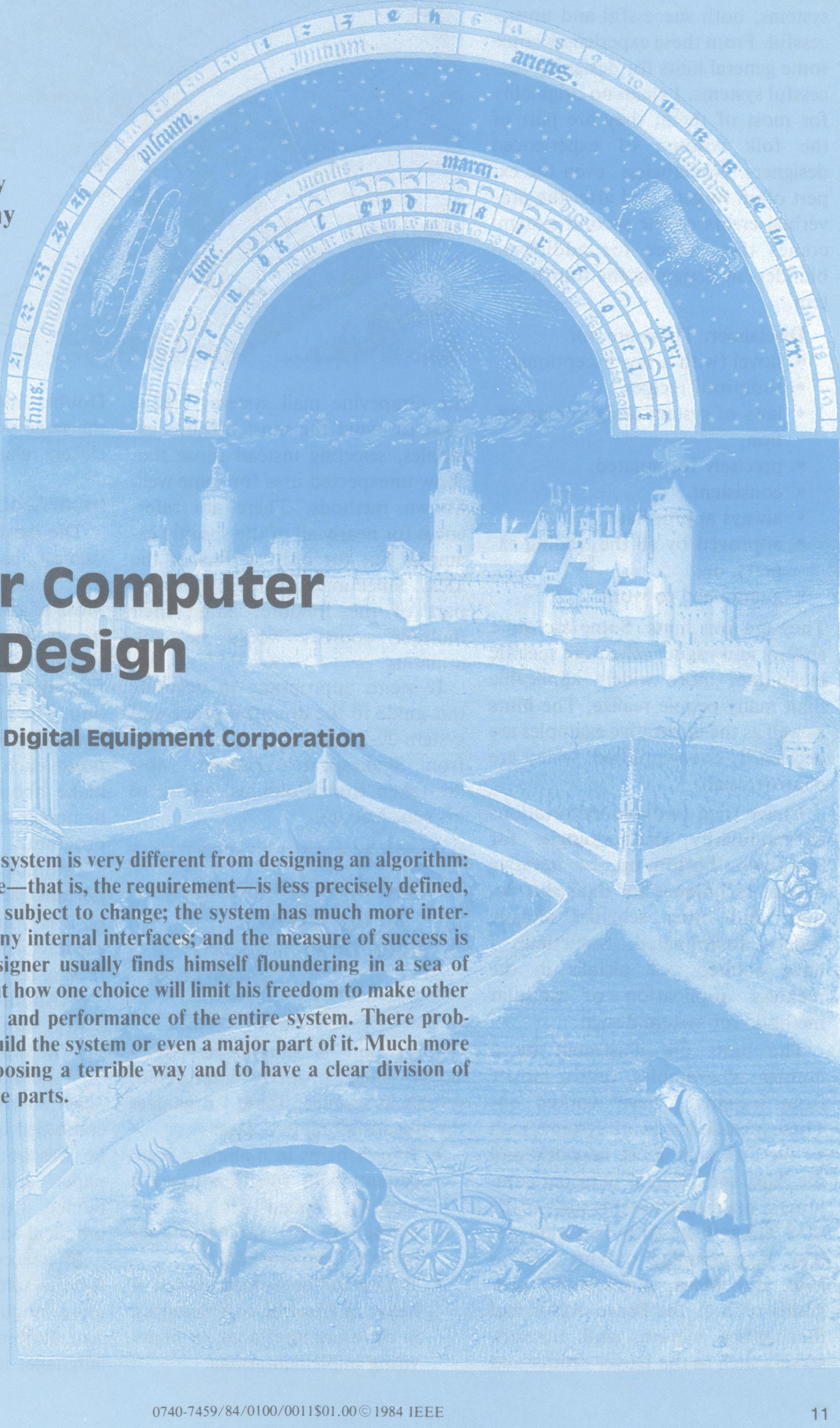
Decorated with pithy quotations from many sources, this collection of good advice and anecdotes draws upon the folk wisdom of experienced designers.



Hints for Computer System Design

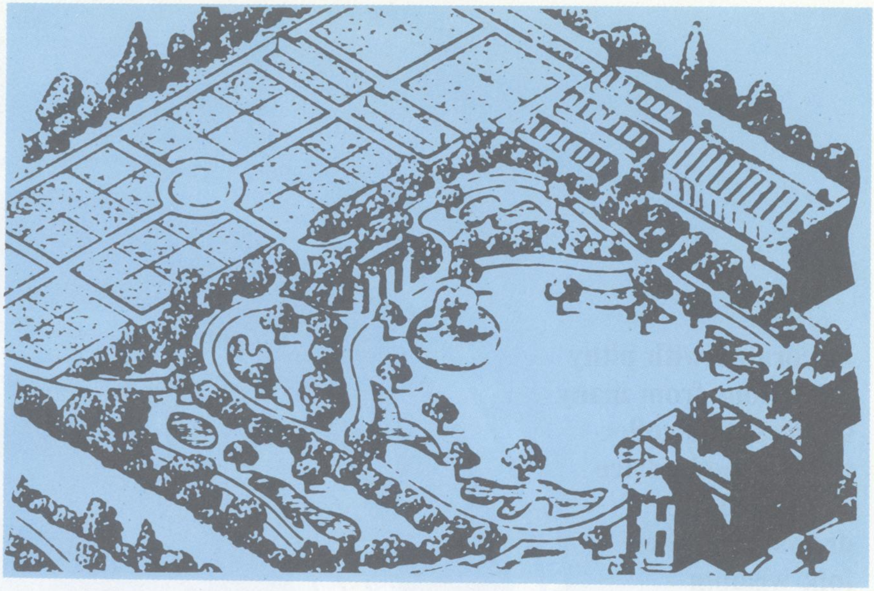
Butler W. Lampson, Digital Equipment Corporation

Designing a computer system is very different from designing an algorithm: the external interface—that is, the requirement—is less precisely defined, more complex, and more subject to change; the system has much more internal structure—hence, many internal interfaces; and the measure of success is much less clear. The designer usually finds himself floundering in a sea of possibilities, unclear about how one choice will limit his freedom to make other choices or affect the size and performance of the entire system. There probably isn't a best way to build the system or even a major part of it. Much more important is to avoid choosing a terrible way and to have a clear division of responsibilities among the parts.

An earlier version of this article appeared in *Proceedings of the Ninth ACM Symposium on Operating Systems Principles,* October 1983, ©1983, Association for Computing Machinery, Inc.

0740-7459/84/0100/0011$01.00© 1984 IEEE

I have designed and built a number of computer systems—some that worked and some that didn't. I have also used and studied many other systems, both successful and unsuccessful. From these experiences come some general hints for designing successful systems. I claim no originality for most of them; they are part of the folk wisdom of experienced designers. Nonetheless, even the expert often forgets, and after the proverbial errors of the second system[1] comes the chance for additional blunders in designing the fourth system.



Disclaimer. These are *not*

- novel (with a few exceptions),
- foolproof recipes,
- laws of system design or operation,
- precisely formulated,
- consistent,
- always appropriate,
- approved by all the leading experts, or
- guaranteed to work.

They are just hints. Some are quite general and vague; others are specific techniques more widely applicable than many people realize. The hints as well as the illustrative examples are necessarily oversimplified. Many are controversial.

I have tried to avoid exhortations to modularity, methodologies for top-down, bottom-up, or iterative design, techniques for data abstraction, and other schemes already widely disseminated. Sometimes I have pointed out pitfalls in the reckless application of popular methods for system design.

The hints are illustrated by a number of examples, drawn mostly from systems I have worked on. These include hardware systems such as the Ethernet local network and the Alto and Dorado personal computers, operating systems such as the SDS 940 and the Alto operating system, programming systems such as Lisp and Mesa, applications programs such as the Bravo editor and Star office system, and network servers such as the Dover printer and the Grapevine mail system. I have tried to avoid the most obvious examples, selecting instead those that show unexpected uses for some well-known methods. There are references for nearly all of the specific examples but for only a few of the ideas. Since many of these ideas are part of the folklore, citing their multiple sources would be voluminous.

It seems appropriate to decorate this guide to the doubtful process of system design with quotations drawn from Shakespeare's *Hamlet*. Most are taken from Polonius' advice to his son, Laertes.

And these few precepts
in thy memory
Look thou character.

—Hamlet,I,iii,58

Each hint is summarized by a slogan that, properly interpreted, reveals its essence. Table 1 organizes the slogans along two axes:

- *Why* it helps in making a good system: with function (does it work?), speed (is it fast enough?), or fault-tolerance (does it keep working?).
- *Where* in the system design it helps: in ensuring completeness, in choosing interfaces, or in devising implementations.

Double lines connect repetitions of the same slogan; single lines connect closely related slogans.

Interfaces

The most important hints, and the vaguest, have to do with obtaining the right function from a system. Most of them depend on the notion of an *interface* separating an *implementation* of some abstraction from the *clients* who use the abstraction. The interface between two programs consists of the set of *assumptions* each programmer needs to make about the other program to demonstrate the correctness of his program.[2]

Defining interfaces is the most important part of system design. Usually, it is also the most difficult, since the interface design must satisfy conflicting requirements:

- An interface should be simple.
- It should be complete.
- It should admit a sufficiently small and fast implementation.

Alas, all too often the assumptions embodied in an interface turn out to be misconceptions. Parnas' classic paper[3] and a more recent one on device interfaces[2] offer excellent practical advice on this subject.

Interfaces are difficult to design for the same reasons that programming languages are difficult to design. Indeed, each interface is a small language; it defines a set of objects and the operations used to manipulate the objects. Concrete syntax is not an issue, but every other aspect of programming language design is present. In light of this observation, many of Hoare's hints on language design[4] can be read as supplementing this article.

Keep it simple

Do one thing at a time, and do it well. An interface should capture the *minimum* essentials of an abstraction.

> **Perfection is reached, not when there is no longer anything to add, but when there is no longer anything to take away.**
>
> **—A. Saint-Exupery**

Don't generalize. Generalizations are generally wrong:

- "We are faced with an insurmountable opportunity." (W. Kelley)
- "If in doubt, leave it out." (Anonymous)
- "Exterminate features." (C. Thacker)
- "KISS: Keep It Simple, Stupid." (Anonymous)

On the other hand:

- "Everything should be made as simple as possible, but no simpler." (A. Einstein)

When an interface undertakes to do too much, the result is a large, slow complicated implementation. An interface is a contract to deliver a certain amount of service. Clients of the interface depend on the functions it provides, usually documented in the interface specification. They also depend on incurring a reasonable cost in time or other scarce resources for using the interface; however, the definition of "reasonable" is not usually documented anywhere. If there are six levels of abstraction and each costs 50 percent more than is "reasonable," the service delivered at the top will miss by more than a factor of 10.

Consequently, service must have a fairly predictable cost, and the interface must not promise more than the implementer knows how to deliver. Especially, it should not promise features only a few clients need, unless the implementer knows how to provide them without penalizing others. A better implementer—or one who comes along in ten years when the problem is better understood—might be able to deliver all of the promised features. But unless the one you have today can do so, it is wise to reduce your aspirations.

For example, PL/I got into serious trouble by attempting to provide consistent meanings for a large number of generic operations across a wide variety of data types. Early implementations tended to handle all of the cases inefficiently. Even with today's optimizing compilers, 15 years later, it is difficult for the programmer to tell what will be fast and what will be slow.[5] A language such as Pascal or C is much easier than PL/I to use because each construct has a roughly constant cost independent of context or arguments. In fact, most of these constructs have about the same cost.

> **Algol 60 was not only an improvement on its predecessors, but also on nearly all its successors.**
>
> **—C. Hoare**

Of course, these observations apply most strongly to interfaces that clients use heavily—virtual memory, files, display handling, and arithmetic. A seldom-used interface—such as password checking, inter-

Table 1.
Summary of the slogans.

	FUNCTIONALITY Does it work?	SPEED Is it fast enough?	FAULT-TOLERANCE Does it keep working?
COMPLETENESS	Separate normal and worst case	Safety first Shed load End-to-end	End-to-end
INTERFACE	Do one thing well: Don't generalize Get it right Don't hide power Use procedure arguments Leave it to the client Keep basic interfaces stable Keep a place to stand	Make it fast Split resources Static analysis Dynamic translation	End-to-end Log updates Make actions atomic
IMPLEMENTATION	Plan to throw one away Keep secrets Use a good idea again Divide and conquer	Cache answers Use hints Use brute force Compute in background Batch processing	Make actions atomic Use hints

preting user commands, or printing 72-point characters—can sacrifice some performance for functionality. (What this really means is that although the cost must still be predictable, it can be many times the achievable minimum.) Of course, such cautious rules don't apply to research whose object is learning how to make better implementations. But since research may fail, others mustn't depend on its success.

Examples of offering too much are legion. The Alto operating system has an ordinary read/write-*n*-bytes interface to files; it was extended for Interlisp-D[7] with an ordinary paging system that stores each virtual page on a dedicated disk page. Both systems have small implementations (about 900 lines of code for files, 500 for paging) and are fast (a page fault takes on disk access and has a constant computing cost that is a small fraction of the disk access time; the client can fairly easily run the disk at full speed).

The Pilot system,[8] which succeeded the Alto OS, followed Multics and several other systems in allowing virtual pages to be mapped to file pages, thus subsuming file I/O within the virtual memory system. The implementation is much larger (about 11,000 lines of code) and slower (it often incurs two disk accesses to handle a page fault and cannot run the disk at full speed). Thus, the extra functionality has a high price.

That is not to say that a good implementation of this interface is impossible, but it is difficult. Pilot was designed and coded by several highly competent and experienced people. Part of the problem is avoiding circularity: the file system would like to use the virtual memory, but virtual memory depends on files. Quite general ways are known to solve this problem,[9] but they are tricky, and easily lead to greater cost and complication in the normal case.

Another example illustrates how easily generality can lead to unexpected complexity. The Tenex system[10] has the following innocent-looking combination of features:

- It reports a reference to an unassigned virtual page by a trap to the user program.
- A system call is viewed as a machine instruction for an extended machine. Any reference it makes to an unassigned virtual page is similarly reported to the user program.
- Large arguments to system calls, including strings, are passed by reference.

And, in this upshot,
purposes mistook
Fall'n on th'
inventors' heads.

—***Hamlet*, V,ii,385**

- A system call CONNECT obtains access to another directory; one of its arguments is a string containing the password for the directory. If the password is wrong, the call fails after a three-second delay, preventing the guessing of passwords at high speed.

CONNECT is implemented by the loop of the form

```
for i := 0 to Length [DirectoryPassword] do
  if DirectoryPassword[i] ≠ PasswordArgument[i] then
        Wait three seconds;
        return BadPassword
Connect to directory; return Success
```

A password of length *n* can be guessed in 64*n* tries on the average, rather than $128^n/2$ (Tenex uses 7-bit characters in strings), by the following trick. Arrange the *PasswordArgument* so that its first character is the last character of a page and the next page is unassigned. Next, try each possible character as the first. If CONNECT reports a *BadPassword,* the guess was wrong; if the system reports a reference to an unassigned page, it was correct. Now arrange the *PasswordArgument* so that its second character is the last character of the page and proceed in the obvious way.

This obscure and amusing bug went unnoticed by the designers because the interface provided by a Tenex system call is quite complex; it includes the possibility of a reported reference to an unassigned page. Looking at it another way, the interface provided by an ordinary memory reference instruction in system code is quite complex; it includes the possibility that an improper reference will be reported to the client, without any chance for the system code to get control first.

At times, however, it's worth a lot of work to make a fast implementation of a clean and powerful interface. If the interface is used widely enough, the effort put into designing and tuning the implementation can pay off many times over. But do this only when its importance is already known from existing uses. And be sure that you know how to make it fast.

For example, Dan Ingalls devised the *BitBlt* or *RasterOp* interface for manipulating raster images[11,12] after several years of experimenting with the Alto's high-resolution interactive display. Its implementation cost about as much microcode as the entire emulator for the Alto's standard instruction set and required a lot of skill and experience to construct. Nevertheless, its performance is nearly as good as the special-purpose character-to-raster operations that preceded it, and its simplicity and generality have made a big difference in the ease of building display applications.

An engineer is a man who can do for a dime what any fool can do for a dollar.

—Anonymous

The Dorado memory system[13] contains a cache and a separate high-bandwidth path for fast input/output. It provides a cache read or write in every 64-ns cycle, together with

500M bits/second of I/O bandwidth, virtual addressing from both cache and I/O, and no special cases for the microprogrammer to worry about. However, the implementation takes 850 MSI chips and consumed several man-years of design time. This could be justified only by extensive prior experience (30 years!) with this interface and by the knowledge that memory access is usually the limiting factor in performance. Even so, in retrospect the high I/O bandwidth does not seem to be worth the cost, since it is used mainly for displays. A dual-ported frame buffer almost certainly would be better.

Get it right. Neither abstraction nor simplicity is a substitute for getting it right. In fact, abstraction can be a source of severe difficulties, as the following cautionary tale shows.

Word processing and office information systems usually provide for embedding named fields in the documents they handle. For example, a form letter might have *address* and *salutation* fields. A document is usually represented as a sequence of characters, and a field is encoded by something like {*name: contents*}. Among other operations, a procedure *FindNamedField* finds the field with a given name.

For some time, one major commercial system used a *FindNamedField* procedure that ran in n^2 steps, where n is the length of the document. This remarkable result was achieved by first writing a procedure *FindIthField* to find the ith field (which must take n steps if there is no auxiliary data structure), and then implementing *FindNamedField* [*name*] with the very natural program

```
for i = 0 to NumberOfFields do
    FindIthField;
  if its name is name then exit
```

Once the (unwisely chosen) abstraction *FindIthField* is available, only a lively awareness of its cost will avoid this disaster. Of course, this is not an argument against abstraction, but it is well to be aware of its dangers.

Corollaries

The hints about simplicity and generalization have many corollaries. The first concerns speed.

Make it fast. If it's fast, rather than general or powerful, one client can program the specific function it needs, and another client can program some other function. It is much better to have basic operations executed quickly than to have more powerful operations executed more slowly. (Of course, a fast, powerful operation is best, if you know how to get it.) The trouble with slow, powerful operations is that the client who doesn't want the power pays more for the basic function. Usually, it turns out that the powerful operation is not the right one.

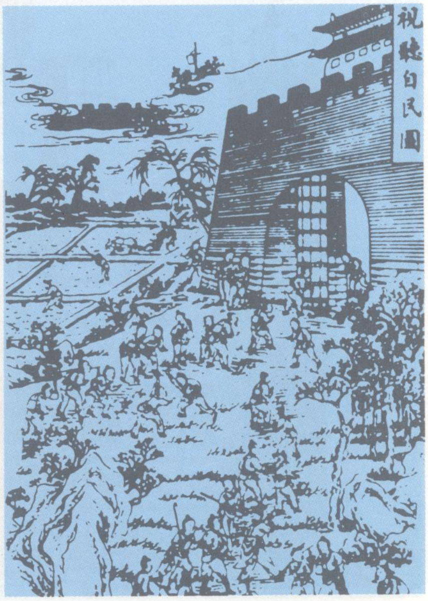
**Costly thy habit as thy purse can buy,
But not expressed in fancy; rich, not gaudy.**
—*Hamlet*, I,iii,70

For example, studies[14-16] have shown that programs spend most of their time doing very simple things—loads, stores, tests for equality, adding one. Machines such as the 801[17] or the RISC,[18] with instructions to do these simple operations quickly, run programs faster for the same amount of hardware than do machines such as the VAX, with more general and powerful instructions that take longer in simple cases. It is easy to lose a factor of two in the running time of a program, with the same amount of hardware in the implementation. Machines with still more grandiose ideas about what the client needs do even worse.[19]

Measurement tools that will pinpoint the time-consuming code are necessary for finding the places where time is being spent in a large system. Few systems are well enough understood to be properly tuned without such tools. It is normal for 80 percent of the time to be spent in 20 percent of the code, but *a priori* analysis or intuition usually can't find the 20 percent with any certainty. The performance tuning of Interlisp-D described by Burton and others[7] illustrates one set of useful tools, giving many details on how the system was sped up by a factor of 10.

Don't hide power. This slogan is closely related to the one about speed. When a low level of abstraction allows something to be done quickly, higher levels should not bury this power inside something more general. The purpose of abstractions is to conceal *undesirable* properties; desirable ones should not be hidden. Sometimes, of course, an abstraction is multiplexing a resource, and this necessarily has some cost. But it should be possible to deliver all or nearly all of it to a single client with only a slight loss of performance.

For example, the Alto disk hardware[20] can transfer a full cylinder at disk speed. The basic file system[6] can transfer successive file pages to client memory at full disk speed, with time for the client to do some computing on each sector so that, with a few sectors of buffering, the entire disk can be scanned at disk speed. This facility has been used to write a variety of applications, ranging from a scavenger that reconstructs a broken file system to programs that search files for substrings matching a pattern.



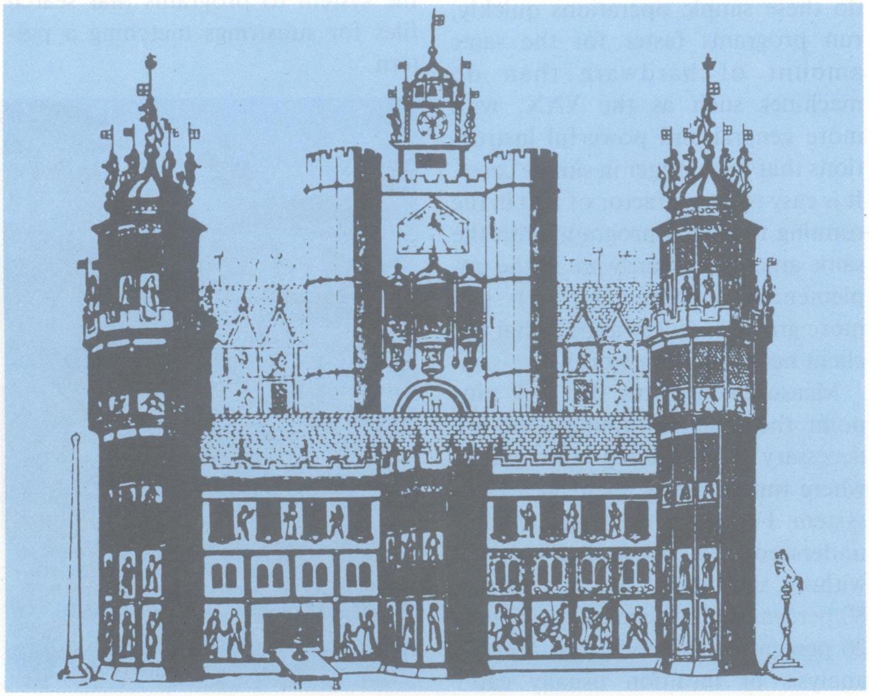
The stream level of the file system can read or write *n* bytes to or from client memory; any portion of the *n* bytes occupying full disk sectors are transferred at full disk speed. Loaders, compilers, editors, and many other programs depend for their performance on this ability to read large files quickly. At this level, the client gives up the facility to see the pages as they arrive—the only price paid for the higher level of abstraction.

Use procedure arguments. They can provide flexibility in an interface. If protection or portability is required, the procedures can be restricted or encoded in various ways. This technique can greatly simplify an interface, eliminating a jumble of parameters that in effect provide a small programming language. A simple example is an enumeration procedure that returns all the elements of a set satisfying some property. The cleanest interface allows the client to pass a filter procedure that tests for the property, rather than defining a special language of patterns or whatever.

This theme has many variations. A more interesting example is the Spy system-monitoring facility in the 940 system at Berkeley.[21] It allows a more or less untrusted user program to plant patches in the code of the supervisor. Although a patch is coded in machine language, the operation that installs it checks that it does no wild branches, contains no loops, is not too long, and stores only in a designated region of memory dedicated to collecting statistics. Using the Spy, the student of the system can fine tune his measurements without fear of breaking the system or perturbing its operation much.

Another unusual example illustrating the power of this method is the FRETURN mechanism in the Cal timesharing system·for the CDC 6400.[22] From any supervisor call *C*, it is possible to make another one, *CF*, that executes exactly like *C* in the normal case but sends control to a designated failure handler if *C* gives an error return. The *CF* operation can do more—for example, it can extend files on a fast, limited-capacity storage device to larger files on a slower device—but it runs as fast as *C* in the (hopefully) normal case.

On the other hand, a specialized language may be better than procedure arguments if it is more amenable to static analysis for optimization, a major criterion in the design of database query languages, for example.



Leave it to the client. As long as it is cheap to pass control back and forth, an interface can combine simplicity, flexibility, and high performance by solving only one problem and leaving the rest to the client. For example, many parsers confine themselves to doing the context-free recognition, calling client-supplied "semantic routines" to record the results of the parse. This approach has obvious advantages over always building a parse tree the client must traverse to find out what happened.

The success of monitors[23,24] as synchronization devices is partly due to the fact that the locking and signaling mechanisms do very little, leaving all the real work to the client programs in the monitor procedures. This simplifies the monitor implementation and keeps it fast. If the client needs buffer allocation, resource accounting, or other frills, the monitor provides these functions or calls other library facilities, paying for what it needs. Although often cited as a drawback, the fact that monitors give no control over the scheduling of processes waiting on monitor locks or condition variables is actually an advantage, since it leaves the client free to provide the scheduling it needs (using a separate condition variable for each class of process), without having to pay for or fight with some built-in mechanism unlikely to do the right thing.

The Unix system[25] encourages the building of small programs that take one or more character streams as input, produce one or more streams as output, and do one operation. When this style is imitated properly, each program has a simple interface and does one thing well, leaving the client to combine a set of such programs with its own code to achieve precisely the desired effect.

Continuity

There is constant tension between the desire to improve a design and the need for stability or continuity.

Keep basic interfaces stable. Since an interface embodies assumptions shared by more than one part of a system, and sometimes by a great many parts, it is very desirable not to change the interface. When the system is programmed in a language without type-checking, it is nearly out of the question to change any public interface, because there is no way of tracking down its clients and checking for elementary incompatibilities, such as disagreements on the number of arguments or confusion between pointers and integers.

However, with a language like Mesa,[26] which has complete type-checking and language support for interfaces, it becomes much easier to change interfaces without causing the system to collapse. But even if type-checking can usually detect that an assumption no longer holds, a programmer still must correct the assumption. When a system has hundreds of thousands of lines of code, the amount of change becomes intolerable; even when there is no doubt about what has to be done, it takes too long to do it. There is no choice but to break the system into smaller pieces related only by interfaces which are stable for years. Traditionally, only the interface defined by a programming language or operating system kernel is this stable.

Keep a place to stand. This is important if you do have to change interfaces. Two examples illustrate this idea. The first is the *compatibility package,* which implements an old interface on top of a new system. This allows programs that depend on the old interface to continue working. Many new operating systems, including Tenex[10] and Cal,[27] have kept old software usable by simulating the supervisor calls of an old system (Tops-10 and Scope, respectively). Usually these simulators need only a small amount of effort compared to the cost of reimplementing the old software, and it is not hard to get acceptable performance.

At a different level, IBM 360/370 systems emulate the instruction sets of older machines, such as the 1401 and 7090. Taken a little further, this approach leads to virtual machines, which simulate several copies of a machine on the machine itself.[28]

A rather different example is the *world-swap debugger.* This debugging system works by writing the real memory of the target system (the one being debugged) onto a secondary storage device, reading in the debugging system in its place. The debugger provides its user with complete access to the target world, mapping each target memory address to the proper place on secondary storage. With some care, it is possible to swap the target back in and continue execution. While this is somewhat clumsy, it allows very low levels of a system to be debugged conveniently, since the debugger does not depend on the correct functioning of anything in the target, except the very simple world-swap mechanism.

This procedure is especially useful during bootstrapping. There are many variations. For instance, the debugger can run on a different machine, with a small *teledebugging* nub in the target world that can interpret *ReadWord, WriteWord, Stop,* and *Go* commands arriving from the debugger over a network. If the target is a process in a timesharing system, the debugger can run in a different process.

Making implementations work

Perfection must be reached by degrees; she requires the slow hand of time.

—Voltaire

Plan to throw one away. You will anyhow.[1] If there is anything new about a system's function, the first implementation will have to be redone completely to be satisfactory—that is, acceptably small, fast, and maintainable. It costs a lot less if you plan to have a prototype.

Unfortunately, two prototypes are needed sometimes, especially if there is a lot of innovation. If you are lucky, you can copy a lot from a previous system; thus, Tenex was based on the SDS 940.[10] This strategy can work even if the previous system was too grandiose; thus, Unix took many ideas from Multics.[25]

Even when an implementation is successful, it still pays to revisit old decisions as the system evolves. In particular, optimizations for particular properties of the load or the environment—memory size, for example—often come to be far from optimal.

An efficient program is an exercise in logical brinkmanship.

—E. Dijkstra

Keep secrets. Secrets are assumptions about an implementation that client programs are not allowed to make.[2] In other words, they are things that can change. The interface defines the things that cannot change without simultaneous changes to both implementation and client. Obviously, it is easier to program and modify a system if its parts make fewer assumptions about each other. On the other hand, the system may not be easier to design—it's difficult to design a good interface. Finally, there is tension between keeping secrets and not hiding power.

There is another danger in keeping secrets. One way of improving performance is *increasing* the number of assumptions one part of a system makes about another. These additional assumptions often allow less work to be done, sometimes a lot less. For instance, if a set of size n is known to be sorted, it is possible to do a membership test in time log n rather than n. This technique is very important in the design of algorithms and the tuning of small modules. In a large system, the ability to improve each part separately is usually more important. Striking the right balance remains an art.

Divide and conquer. A well-known method for solving a large, difficult problem is to divide it into several smaller and easier ones. The resulting program is usually recursive. When resources are limited, the method takes a slightly different form—bite off as much as will fit, leaving the rest for another iteration.

> **O, throw away the**
> **worser part of it,**
> **And live the purer**
> **with the other half.**
>
> —***Hamlet,*** **III,iv,158**

A good example is the Alto's Scavenger program that scans the disk and rebuilds the index and directory structures of the file system from the file identifier and page number recorded on each disk sector.[6] A recent rewrite of this program has a phase in which it builds a data structure in main storage, with one entry for each contiguous run of disk pages that is also a contiguous set of pages in a file. Normally, files are allocated more or less contiguously, and this structure is not too large.

If the disk is badly fragmented, however, the structure will not fit in storage. When this happens, the Scavenger discards the information for half the files and continues with the other half. After the index for these files is rebuilt, the process is repeated for the other files. If necessary, the work is further subdivided. The method fails only if a single file's index won't fit.

Another interesting example is the Dover raster printer.[29,20] It scan-converts lists of characters and rectangles into a large $m \times n$ array of bits, in which 1s correspond to spots of ink on the paper and 0s to spots without ink. In this printer, m = 3300 and n = 4200, so the array contains fourteen million bits and is too large to store in memory. Since the printer consumes bits faster than the available disks can deliver them, the array cannot be stored on disk. Instead, the printer electronics contains two *band buffers,* each of which can store 16 × 4200 bits. The entire array is divided into 16 × 4200 bit slices called *bands.* The characters and rectangles are sorted into *buckets,* one for each band. A bucket receives the objects that start in the corresponding band.

Scan conversion proceeds by filling one band buffer from its bucket, and then playing it out to the printer and zeroing it while the other buffer is filled from the next bucket. Objects spilling over the edge of one band are put on a *left-over list,* which is merged with the contents of the next bucket. The left-over scheme thus allows the problem to be subdivided.

Sometimes it is convenient to artificially limit the resource by *quantizing* it in fixed-size units, a method that simplifies bookkeeping and prevents one kind of fragmentation. The classic example is using fixed-size pages, rather than variable-size segments, for virtual memory. In spite of the apparent advantages of keeping logically related information together and transferring it between main storage and backing storage as a unit, paging systems have worked out better. The reasons for this are complex and have not been systematically studied.

> **Rather bear those ills we have**
> **Than fly to others that**
> **we know not of.**
>
> —***Hamlet,*** **III,i,28**

Use a good idea again. Don't generalize. A specialized implementation of an idea may be much more effective than a general one. An interesting example is the notion of replication of data for reliability. A small amount of data can easily be replicated locally by writing it on two or more disk drives.[30] When the amount of data is large or the data must be recorded on separate machines, it is not easy to ensure that the copies are always the same.

Gifford[31] shows how the problem can be solved by building replicated data on top of a transactional storage system, allowing an aribtrarily large update to be done as an atomic operation. The transactional storage depends on the simple local replication scheme to store its log reliably. There is no circularity since only the idea is used twice, not the code. A third possible use of replication in this context is to store the commit record on several machines.[32]

The user interface for the Star office system[33] has a small set of operations—type text, move, copy, delete, show properties—that are applied to nearly all of the objects in the system, including text, graphics, file folders and file drawers, record files, printers, and in and out baskets. The exact meaning of an operation varies with the class of object, within the limits of what the user is likely to consider natural. For instance, copying a document to an out basket causes it to be sent as a message; moving the endpoint of a line causes the line to follow like a rubber band. Certainly, the implementations are quite different in many cases. But the generic operations do not simply make the system easier to use; they represent a view of what operations are possible and how the implementation of each class of object should be organized.

Handle normal and worst cases separately. The requirements for the two are quite different: the normal case must be fast, and the worst case must make some progress.

In most systems it is all right to schedule unfairly, giving no service to some process, or to deadlock the entire system, as long as this event is detected automatically and doesn't happen too often. The usual recovery is by crashing some process or even the entire system. At first, this sounds terrible, but one crash per week is usually a cheap price to pay for 20 percent better performance. Of course, the system must have decent error recovery (an application of the end-to-end principle discussed later), but that is required in any case, since there are so many other possible causes of a crash.

Caches and hints are examples of special treatment for the normal case, but there are many others.

Diseases desperate grown
By desperate appliance
are relieved,
Or not at all.

—*Hamlet*, IV,iii,9

Therefore this project
Should have a back or
second, that might hold
If this did blast in proof.

—*Hamlet*, IV,vii,152

The Interlisp-D and Cedar programming systems use a reference-counting garbage collector[34] that has an important optimization of this kind. Pointers in the local frames or activation records of procedures are not counted; instead, the frames are scanned whenever garbage is collected, saving a lot of reference counting since most pointer assignments are to local variables. Since there are not many frames, the time to scan them is small, and the collector is nearly real time.

Cedar goes further by not keeping track of which local variables contain pointers and assuming they all do. Therefore, an integer that contains the address of an object no longer referenced will prevent that object from being freed. Measurements show that less than one percent of the storage is incorrectly retained.[35]

Reference counting makes it easy to have an incremental collector so that computation need not stop during collection. However, it cannot reclaim circular structures that are no longer reachable. Therefore, Cedar has a conventional trace-and-sweep collector as well. Such a collector is not suitable for real-time applications, since it stops the entire system for many seconds, but in interactive applications it can be used during coffee breaks to reclaim accumulated circular structures.

Another problem with reference counting is that the count may overflow the space provided for it. This seldom happens, since only a few objects have more than two or three references. It is simple to make the maximum value sticky. Unfortunately, in some applications, the root of a large structure is referenced from many places; if the root's count becomes sticky, a lot of storage will unexpectedly become permanent.

An attractive solution is to have an *overflow count* table—a hash table keyed on the address of an object. When the count reaches its limit, it is reduced by half, the overflow count is increased by one, and the overflow flag is set in the object. When the count reaches zero, the process is reversed if the overflow flag is set. Thus, even with as few as four bits, there is room to count up to seven; the overflow table is touched only when the count swings by more than four, which seldom happens.

In many cases, resources are dynamically allocated and freed (e.g., real memory in a paging system), and additional resources sometimes are needed temporarily to free an item (some table might have to be swapped in to discover where to write out a page). There is normally a cushion (clean pages that can be freed with no work), but in the worst case the cushion may disappear (all pages are dirty).

The trick here is to keep a little something in reserve under a mattress, only bringing it out in a crisis. Bounding the resources needed to free one item determines the size of the reserve under the mattress, which must be regarded as a fixed cost of the resource multiplexing. When the crisis arrives, only one item should be freed at a time so that the entire reserve is devoted to that job. While this may slow things down a lot, it ensures that progress will be made.

Sometimes, radically different strategies are appropriate in the normal and worst cases. The Bravo editor[36] uses a *piece table* to represent the document being edited. This is an array of *pieces*—pointers to strings of characters stored in a file. Each piece contains the file address of the first character in the string and its length. The strings are never modified during normal editing. Instead, when characters are deleted, the piece containing the deleted characters is split into two pieces, one pointing to the first undeleted string and the other to the second. When characters are inserted from the keyboard, they are appended to the file. The piece containing the insertion point is split into three pieces: one for the preceding characters, a second for the inserted characters, and a third for the following characters.

After hours of editing, when there are hundreds of pieces and things start to bog down, it is time for a *cleanup,* which writes a new file containing all the characters of the document in order. Afterwards, the piece table can be replaced by a single piece pointing to the new file, and editing can continue. Cleanup is a specialized kind of garbage collection; it can be done in background to avoid interrupting the editing service, although Bravo does not do this.

Speed

The discussion below gives hints for making systems faster. Bentley's excellent book[37] expands some of these ideas and offers many others.

Neither a borrower nor
a lender be,
For loan often loses
both itself and friend,
And borrowing dulleth
edge of husbandry.

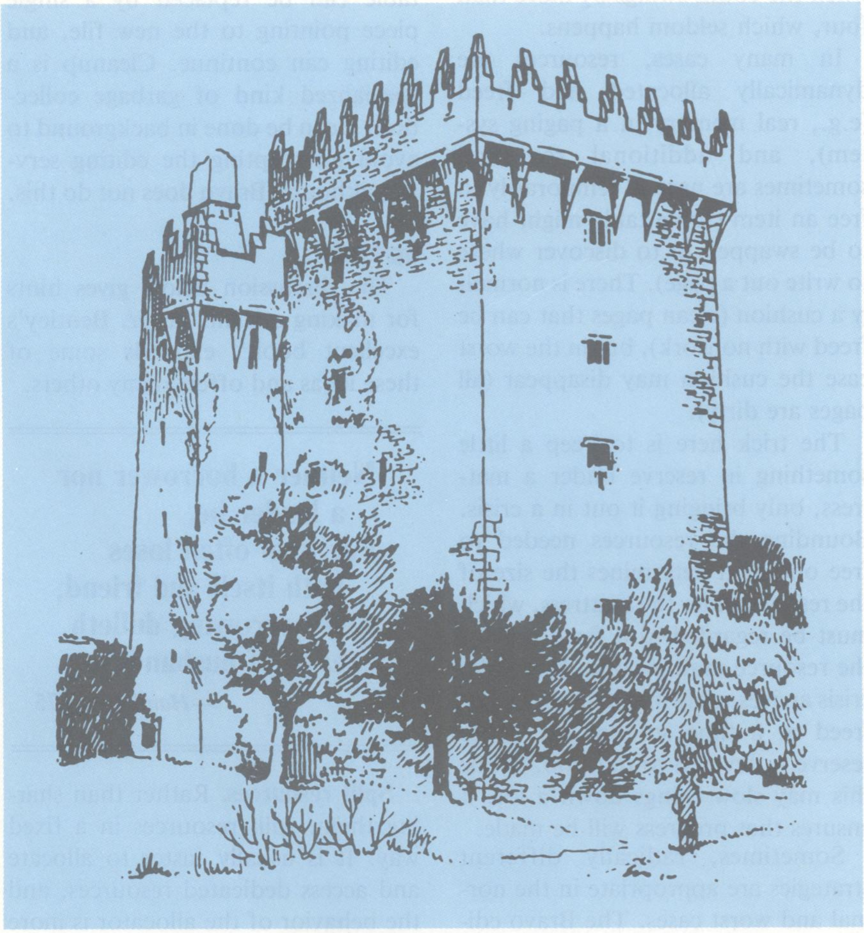
—*Hamlet*, I,iii,75

Split resources. Rather than sharing them, split resources in a fixed way. It is usually faster to allocate and access dedicated resources, and the behavior of the allocator is more predictable. The obvious disadvantage is that more total resources are needed, ignoring multiplexing overheads, than if all come from a common pool. In many cases, however, the cost of the extra resources is small or the overhead is larger than the fragmentation, or both.

For example, it is always faster to access information in the registers of a processor than to get it from memory, even if the machine has a high-performance cache. Registers have gotten a bad name for two reasons: it can be tricky to allocate them intelligently, and saving and restoring them across procedure calls may negate their speed advantages. Nevertheless, when programs are written in the approved modern style, with lots of small procedures, 16 registers are nearly always enough for all local variables and temporaries. Thus, there are usually enough registers, and allocation is not a problem. With *n* sets of registers arranged in a stack, saving is needed only when there are *n* successive calls without a return.[38,18]

I/O channels, floating-point coprocessors, and similar specialized computing devices are other applications of this principle. When extra hardware is expensive, these services are provided by multiplexing a single processor. As it becomes cheaper, static allocation of computing power for various purposes becomes worthwhile.

The Interlisp virtual memory system mentioned earlier[7] needs to keep track of the disk address corresponding to each virtual address. This information could be held in the virtual memory, as it is in several systems including Pilot,[8] but the need to avoid circularity makes this rather complicated. Instead, real memory is dedicated. Unless the disk is ridiculously fragmented, the space consumed is less than the space for the code needed to prevent circularity.



Use static analysis. This is a generalization of the last slogan. The result of static analysis is known properties of the program that often can be used to improve its performance. The "if you can" is the hooker; when a good static analysis is not possible, don't delude yourself with a bad one; instead, fall back on a dynamic scheme.

The remarks earlier about registers depend on the fact that the compiler can decide how to allocate them, simply by putting the local variables and temporaries there. Most machines lack multiple sets of registers or lack a way of stacking them efficiently. Good allocation is then much more difficult, requiring an elaborate interprocedural analysis that may not succeed and must be redone each time the program changes. So a little bit of dynamic analysis (stacking the registers) goes a long way. Of course, the static analysis can still pay off in a large procedure if the compiler is clever.

A program can read data much faster when the data is read sequentially. This makes it easy to predict what data will be needed next and read it ahead into a buffer. Often the data can be allocated sequentially on a disk, allowing it to be transferred at least an order of magnitude faster. These performance gains depend on the fact that the programmer has arranged the data so that it is accessed according to some predictable pattern—that is, so that static analysis is possible. Although many attempts have been made to analyze programs after the fact and optimize the disk transfers, as far as I know they have never worked. The dynamic analysis done by demand paging is always at least as good.

Some kinds of static analysis exploit the fact that some invariant is maintained. A system that depends on such facts may be less robust in the face of hardware failures or bugs in the software that falsify the invariant.

Dynamic translation. An important variation on the old idea of compiling, dynamic translation converts a convenient (compact, easily modified, or easily displayed) representation to one that can be quickly interpreted. Translating a bit at a time is the idea behind separate compilation, which goes back at least to Fortran II. Incremental compilers do it automatically when a statement or procedure is changed.

Mitchell investigated smooth motion on a continuum between the convenient and the fast representation.[39] A simpler version of his scheme is to always do the translation on demand and cache the result; then, only one interpreter is required, and no decisions are needed except for cache replacement.

For example, an experimental Smalltalk implementation[40] uses the byte codes produced by the standard Smalltalk compiler as the convenient (in this case, compact) representation, and translates a single procedure from byte codes into machine language when it is invoked. It keeps a cache with room for a few thousand instructions of translated code. For the scheme to pay off, the cache must be large enough so that a procedure is executed at least *n* times, where *n* is the ratio of translation time to execution time for the untranslated code.

The C-machine stack cache[38] provides a rather different example. In this device, instructions are fetched into an instruction cache; as they are loaded, any operand address relative to the local frame pointer *FP* is converted into an absolute address by using the current value of *FP,* which remains constant during execution of the procedure. In addition, if the resulting address is in the range of addresses currently in the stack data cache, the operand is changed to register mode; later execution of the instruction will then access the register directly in the data cache. The *FP* value is concatenated with the instruction address to form the key of the translated instruction in the instruction cache so that multiple activations of the same procedure will still work.

Cache answers. Cache answers to expensive computations, rather than doing them over. By writing the triple $[f,x,f(x)]$ in an associative store with f and x as keys, we can retrieve $f(x)$ with a lookup. This wins if $f(x)$ is needed again before it gets replaced in the cache, which presumably has limited capacity. How much it wins depends on how expensive it is to compute $f(x)$.

**If thou didst ever
hold me in thy heart.**

—*Hamlet,* V,ii,345

A serious problem is that if f is not functional (can give different results with the same arguments), we need a way to *invalidate* or *update* a cache entry when the value of $f(x)$ changes. Updating depends on an equation of the form $f(x+\Delta) = g(x,\Delta,f(x))$ in which g is much cheaper to compute than f. For example, x might be an array of 1000 numbers, $f(x)$ the sum of the array elements, and Δ a new value for x_i. Then $g(x,\Delta,sum)$ is $sum - x_i + \Delta$.

If a cache is too small to hold all the "active" values, it will thrash. If recomputing f is expensive, performance will suffer badly. Thus, it is wise to choose the cache size adaptively, if possible, increasing it when the hit rate declines and reducing it when many entries go unused for a long time.

The classic example is a hardware cache speeding up access to main storage. Its entries are triples [*Fetch, address, contents of address*]. The *Fetch* operation is certainly not functional: *Fetch*(*x*) gives a different answer after *Store*(*x*) has been done. Hence, the cache must be updated or invalidated after a store. Virtual memory systems do exactly the same thing: main storage plays the role of cache, disk plays the role of main storage, and the unit of transfer is the page, segment, or whatever.

But nearly every nontrivial system has more specialized applications of caching. This is especially true for interactive or real-time systems in which the basic problem is to incrementally update a complex state in response to frequent small changes. Doing this in an ad-hoc way is extremely error-prone. The best organizing principle is to recompute the entire state after each change, caching the expensive results of this computation. A change must invalidate at least the cache entries it renders invalid. If these are too difficult to identify precisely, more entries may be invalidated—at the price of more computing to reestablish them. The secret of success is to organize the cache so that small changes invalidate only a few entries.

For example, the Bravo editor[36] has a function *DisplayLine*[*document, characterNumber*] that returns the bitmap for the line of text in the displayed document with *document*[*characterNumber*] as its first character. It also returns *lastCharDisplayed* and *lastCharUsed,* the numbers of the last character displayed and the last character examined in computing the bitmap. These are usually not the same, since it is necessary to look past the end of the line to choose the line break. This function computes line breaks, does justification, uses font tables to map characters into their raster pictures, etc.

There is a cache with an entry for each line currently displayed on the screen and sometimes a few lines just above or below. An edit changing characters *i* through *j* invalidates any cache entry for which [*characterNumber* .. *lastCharUsed*] intersects [*i* .. *j*]. The display is recomputed by

```
c: = firstChar;
do
    [bitMap,lastC,]: =
        DisplayLine[document,c];
    Paint[bitMap];
    c: = lastC + 1
od
```

The call of *DisplayLine* is short-circuited by using the cache entry for [*document,c*] if it exists. At the end, any cache entry not used is discarded. These entries are not invalid, but they are no longer interesting, because the line breaks have changed so that a line no longer begins at these points.

The same idea can be applied in a very different setting. Bravo allows a document to be structured into paragraphs, each with specified left and right margins, interline leading, etc. In ordinary page layout, all information about the paragraph necessary for doing the layout can be represented very compactly:

- number of lines,
- height of each line (normally, all lines are the same height),
- any keep properties, and
- pre- and post-leading.

In the usual case, this information can be encoded in three or four bytes. Since a 30-page chapter has perhaps 300 paragraphs, about 1K bytes are required for this data—less information than required to specify the characters on a page. Because the layout computation is comparable to the line layout computation for a page, it should be possible to do the pagination for this chapter in less time than required to render one page. Layout can be done independently for each chapter.

A cache of [*paragraph, ParagraphShape*(*paragraph*)] entries makes this work. If the *paragraph* is edited, the cache entry is invalid and must be recomputed. This can be done at the time of the edit (reasonable if the paragraph is on the screen, as is usually the case, but not so good for a global substitute), in background, or only when repagination is requested. Unfortunately, Bravo does not implement this idea.

Use hints. This helps to speed up normal execution. A hint, like a cache entry, is the saved result of some computation. It is different in two ways from a cache entry: it may be wrong, and it is not necessarily reached by an associative lookup. Since a hint may be wrong, there must be a way to check its correctness before taking any unrecoverable action. It is checked against the *truth*—information that must be correct but can be optimized for this purpose, since it need not be adequate for efficient execution. Like a cache entry, the purpose of a hint is to make the system run faster. Usually this means that it must be correct nearly all the time.

For example, in the Alto[6] and Pilot[8] operating systems, each file has a unique identifier, and each disk page has a *label* field whose contents can be checked before reading or writing the data, without slowing down the data transfer. The label contains the identifier of the file containing the page as well as the number of that page in the file. Page zero of each file is called the *leader* page, containing among other things the directory in which the file resides and its string name in that directory. This is the truth on which the file systems are based, and they take great pains to keep it correct.

The apparel oft proclaims the man.

***—Hamlet,* I,iii,72**

With only this information, however, there is no way to find the identifier of a file from its name in a directory or to find the disk address of page *i*, except to search the entire disk—a method that works but is unacceptably slow. Therefore, each system maintains hints to speed up these operations. Each directory has a file containing triples [string name, file identifier, address of first page]. Each file has a data structure that maps a page number into the disk address of the page.

In the Alto system, this structure is a link in each label to the next label, making it fast to go from page n to page $n + 1$. In Pilot, this structure is a B-tree that implements the map directly, taking advantage of the common case in which consecutive file pages occupy consecutive disk pages. Information obtained from any of these hints is checked when it is used, by checking the label or reading the file name from the leader page. If it proves to be wrong, all of it can be reconstructed by scanning the disk. Similarly, the bit table keeping tracking of free disk pages is a hint. The truth is represented by a special value in the label of a free page that is checked when the page is allocated and before the label is overwritten with a file identifier and page number.

Another example of hints is the store and forward routing first used in the Arpanet.[41] Each node in the network keeps a table giving the best route to each of the other nodes. This table is updated by periodic broadcasts in which each node announces to all the other nodes its opinion about the quality of its links to its nearest neighbors. Since these broadcast messages are neither synchronized nor guaranteed delivery, the nodes may not have a consistent view at any instant. The truth in this case is that each node knows its own identity and hence knows when it receives a packet destined for itself. For the rest, the routing does the best it can. When things aren't changing too fast, it is nearly optimal.

A more curious example is the Ethernet[42] where lack of carrier on the cable is a hint that a packet can be sent. If two senders take the hint simultaneously, there is a *collision* that both can detect. Both stop, delay for a randomly chosen interval, and try again. If n successive collisions occur, it is taken as a hint that the number of senders is 2^n. Each sender then lengthens the mean of his random delay interval accordingly to ensure that the net does not become overloaded.

A very different application of hints speeds up execution of Smalltalk programs.[40] In Smalltalk, the code executed when a procedure is called is determined dynamically, based on the type of the first argument. Thus *Print*[*x,format*] invokes the *Print* procedure that is part of the type of x. Since Smalltalk has no declarations, the type of x is not known statically. Instead, each object has a pointer to a table containing a set of pairs [procedure name, address of code]. When this call is executed, *Print* is looked up in the table for *x*. (I have normalized the unusual Smalltalk terminology and syntax and oversimplified a bit.) This is expensive.

Since it usually turns out that the type of *x* is the same as it was last time, the code for the call *Print*[*x*, *format*] can be arranged like this:

```
push format; push x;
push lastType; call lastProc,
```

and each *Print* procedure begins with

```
lt: = Pop[]; x: = Pop[]; t: = type of x;
if t≠lt then LookupAndCall
        [x,"Print"]
else the body of the procedure.
```

Here *lastType* and *lastProc* are immediate values stored in the code. The idea is that *LookupAndCall* should store the type of *x* and the code address it finds back into the *lastType* and *lastProc* fields, respectively. If the type is the same next time, the procedure will be called directly.

Measurements show that this cache hits about 96 percent of the time. In a machine with an instruction fetch unit, this scheme has the nice property that the transfer to *lastProc* can proceed at full speed. Thus, when the hint is correct, the call is as fast as an ordinary subroutine call. The check of *t*≠*lt* can be arranged so that it normally does not branch.

The same idea in a different guise is used in the S-1,[43] which has an extra bit for each instruction in its instruction cache. The bit is cleared when the instruction is loaded, set when the instruction causes a branch to be taken, and used to govern the path the instruction fetch unit follows. If the prediction turns out to be wrong, the bit is changed and the other path is followed.

When in doubt, use brute force. Especially as the cost of hardware declines, a straightforward, easily analyzed solution that requires a lot of special-purpose computing cycles is better than a complex, poorly characterized solution that may work well if certain assumptions are satisfied. For example, Ken Thompson's Belle chess machine relies mainly on special-purpose hardware, rather than sophisticated chess strategies, to generate moves and evaluate positions. Belle has won the world computer chess championship several times.

Another instructive example is the success of personal computers over timesharing systems. While timesharing systems include more cleverness and have fewer wasted cycles, personal computers are being increasingly recognized as the most cost-effective way of providing interactive computing.

Even an asymptotically faster algorithm is not necessarily better. While it is known how to multiply two $n \times n$ matrices faster than $O(n^{2.5})$, the constant factor is prohibitive. On a more mundane note, the 7040 Watfor compiler uses linear search to look up symbols; student programs have so few symbols that the setup time for a better algorithm couldn't be recovered.

Compute in background, when possible. In an interactive or real-time system, it is good to do as little work as possible before responding to a request. The reason is twofold:

- A rapid response is better for the users.
- The load usually varies a great deal, so there is likely to be idle processor time later, which is wasted unless there is background work to do.

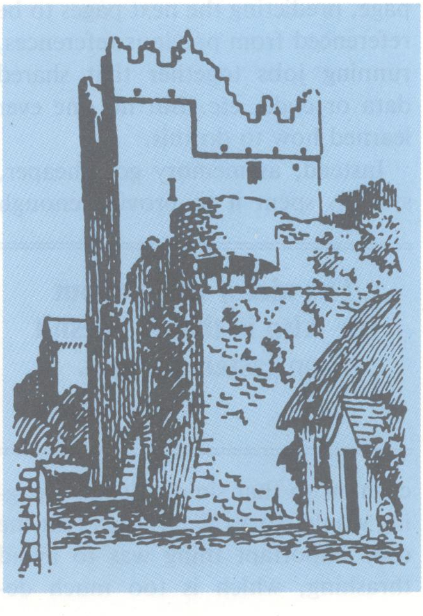
Many kinds of work can be deferred to background. The Interlisp and Cedar garbage collectors[7,34] do nearly all their work this way. Many paging systems write out dirty pages and prepare candidates for replacement in background. Electronic mail can be delivered and retrieved by background processes, since delivery within an hour or two is usually acceptable. Many banking systems consolidate the data on accounts at night and have it ready the next morning.

These four examples have successively less need for synchronization between foreground and background tasks. As the amount of synchronization increases, more care is needed to avoid subtle errors. An extreme example is on-the-fly garbage collection.[44] In most cases, however, a simple producer-consumer relationship is possible between two otherwise independent processes.

Use batch processing, if possible. Doing things incrementally almost always costs more. The fact that disks and tapes work much better when accessed sequentially is only one of many reasons for this. Also, batch processing permits much simpler error recovery.

For example, the Bank of America has an interactive system allowing tellers to record deposits and check withdrawals. Loaded with current account balances in the morning, the system does its best to maintain balances during the day. The next morning, however, the on-line data is discarded and replaced with the results of the previous night's batch run. This design makes it much easier to meet the bank's requirements for trustworthy long-term data, with no significant loss in function.

Safety first. In allocating resources, strive to avoid disaster rather than to attain an optimum.



Many years of experience with virtual memory, networks, disk allocation, database layout, and other resource allocation problems have made it clear that a general-purpose system cannot optimize the use of resources.

Be wary then; best safety lies in fear.

—Hamlet, **I,iii,43**

On the other hand, it is easy enough to overload a system and drastically degrade the service.

A system cannot be expected to function well if the demand for any resource exceeds two-thirds of the capacity, unless the load can be characterized extremely well. Fortunately, hardware is cheap and getting cheaper. We can afford to provide excess capacity. Memory is especially cheap, which is especially fortunate since to some extent plenty of memory can allow other resources, such as processor cycles or communication bandwidth, to be utilized more fully.

The sad truth about optimization was brought home when the first paging systems began to thrash. In those days, memory was very expensive. People had visions of squeezing the most out of every byte by clever optimization of the swapping—putting related procedures on the same page, predicting the next pages to be referenced from previous references, running jobs together that shared data or code, etc. But no one ever learned how to do this.

Instead, as memory got cheaper, systems spent it to provide enough cushion so that simple demand paging would work. We learned that the only important thing was to avoid thrashing, which is too much demand for the available memory. A system that thrashes spends *all* of its time waiting for the disk.

The nicest thing about the Alto is that it doesn't run faster at night.

—J. Morris

The only systems in which cleverness has worked are those with very well-known loads. For instance, the 360/50 APL system[45] had the same size of workspace for each user and common system code for all of them. It made all of the system code resident, allocated a contiguous piece of disk for each user, and overlapped a swap-out and a swap-in with each unit of computation. This was very successful.

We learned a similar lesson about processor time. In interactive use, the response time to a demand for computing is important, since a person is waiting for it. Many attempts were made to tune the processor scheduling as a function of priority of the computation, working set size, memory loading, past history, likelihood of an I/O request, etc. These efforts failed.

Only the crudest parameters produce intelligible effects—for example, interactive vs. noninteractive computations and high, foreground, and background priority. The most successful schemes give a fixed share of the cycles to each job and don't allocate more than 100 percent. Unused cycles are wasted or, with luck, consumed by a background job. The natural extension of this strategy is the personal computer, in which each user has at least one processor to himself.

Shed load. This is a better way to control demand than allowing the system to become overloaded. There are many ways to shed load. An interactive system can refuse new users or, if necessary, deny service to existing users. A memory manager can limit the jobs being served so that their total working sets are less than the available memory. A network can discard packets. If it comes to the worst, the system can crash and start over, hopefully with greater prudence.

Bob Morris once suggested that each terminal in a shared interactive system should have a large red button the user pushes if he is dissatisfied with the service. When the button is pushed, the system must either improve the service or throw the user off, making an equitable choice over a sufficiently long period. The idea is to keep people from wasting their time in front of terminals that are not delivering a useful amount of service.

Give every man thine ear,
but few thy voice; ,
Take each man's censure,
but reserve thy judgment.

—Hamlet, **I,iii,68**

The original specification for the Arpanet[41] was that a packet, once accepted by the net, was guaranteed to be delivered unless the recipient machine was down or a network node failed while it was holding the packet. This turned out to be a bad idea. In the worse case it was very hard to avoid deadlock with this rule; even in the normal case, attempts to obey it led to many complications and inefficiencies. Furthermore, the client did not benefit, since he still had to deal with packets lost by host or network failure. Eventually, the rule was abandoned. The Pup internet,[46] faced with a much more variable set of transport facilities, has always ruthlessly discarded packets at the first sign of congestion.

Fault tolerance

Making a system reliable is not really hard, if you know how to go about it. But retrofitting reliability to an existing design is very difficult.

The unavoidable price of reliability is simplicity.

—C. Hoare

End-to-end. Error recovery done at the application level is absolutely necessary for a reliable system. *Any* other error detection or recovery is *not* logically necessary and is strictly for performance.[47] This observation is very widely applicable.

For example, consider the operation of transferring a file from a file system on a disk attached to machine *A* to another file system on another disk attached to machine *B*. The minimum procedure for inspiring any confidence that the right bits are really on *B*'s disk is to read the file from *B*'s disk, compute a checksum of reasonable length (say, 64 bits), and find that it is equal to a checksum computed by reading the bits from *A*'s disk.

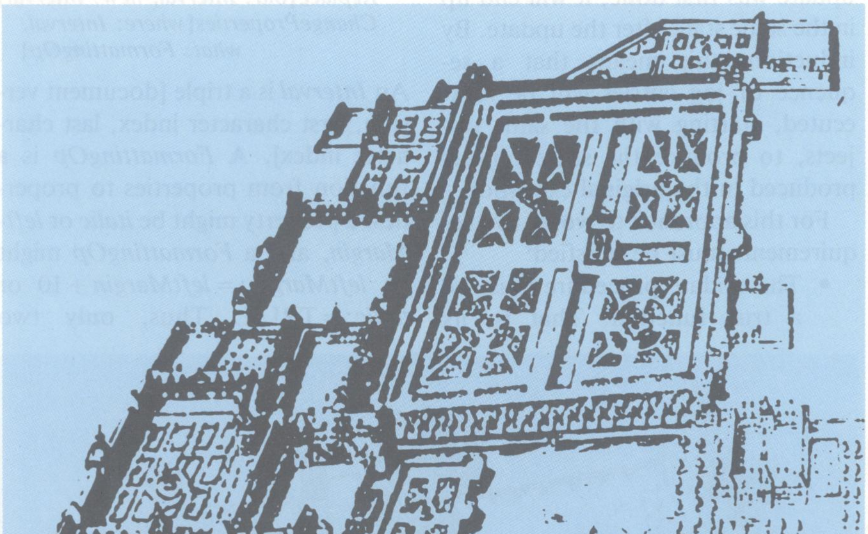
Checking the transfer from *A*'s disk to *A*'s memory, from *A* over the network to *B*, or from *B*'s memory to *B*'s disk is not *sufficient,* since there might be trouble at some other point or the bits might be clobbered while sitting in memory. These other checks are not *necessary* either, because if the end-to-end check fails, the entire transfer can be repeated. Of course, this is a lot of work. If errors are frequent, intermediate checks can reduce the amount of work that must be repeated. However, this is strictly a question of performance, irrelevant to the reliability of the file transfer.

In the ring-based system at Cambridge, it is customary to copy an entire disk pack of 58M bytes with only an end-to-end check. Errors are so infrequent that the 20 minutes of work very seldom needs to be repeated.[48]

Many uses of hints are applications of this idea. For example, in the Alto file system described earlier, it is the check of the label on a disk sector before writing the sector that ensures the disk address for the write is correct. Any precautions taken to make it more likely that the address is correct may be important or even critical for performance, but they do not affect the reliability of the file system.

The Pup internet[46] adopts the end-to-end strategy at several levels. The network's main service is to transport a data packet from a source to a destination. The packet may traverse a number of networks with widely varying error rates and other properties. Internet nodes storing and forwarding packets may run short of space and be forced to discard packets. Only rough estimates of the best route for a packet are available, and these may be wildly erroneous when parts of the network fail or resume operation.

In the face of these uncertainties, the Pup internet provides good service with a simple implementation by attempting only "best efforts" delivery. A packet may be lost with no notice to the sender, or it may be corrupted in transit. Clients must provide their own error control to deal with these problems, and indeed, higher-level Pup protocols do provide more complex services, such as reliable byte streams. The packet transport attempts to report problems to its clients, for example, by providing a modest amount of error control (a 16-bit checksum) and by notifying senders of discarded packets when possible. These services are intended to improve performance in the face of unreliable communication and overloading. Since they are also best efforts, they don't complicate the implementation much.



There are two problems, however, with the end-to-end strategy. First, it requires a cheap test for success. Second, it can lead to working systems with severe performance defects that may not appear until the system becomes operational and is placed under heavy load.

Remember thee?
Yea, from the table
of my memory
I'll wipe away all
trivial fond records,
All saws of books, all
forms, all pleasures past
That youth and
observation copied there,
And thy commandment
all alone shall live
Within the book and
volume of my brain,
Unmixed with baser matter.

—*Hamlet,* I,v,97

Log updates. Record the truth about the state of an object by writing all the updates in a log. A log is a very simple data structure that can be reliably written, reliably read, and cheaply forced out onto disk or other stable storage that can survive a crash. Because the log is append-only, the amount of writing is minimized, and it is easy to ensure the log is valid, no matter when a crash occurs. It is easy and cheap to duplicate the log and write copies on tape.

Logs have been used for many years to ensure that information in a database is not lost,[49] but the idea is a very general one that can be used in ordinary file systems[50,51] and in many other less obvious situations. When a log holds the truth, the current state of the object is very much like a hint (it isn't exactly a hint because there isn't a cheap way to check its correctness).

To use the technique, record every update to an object as a log entry that consists of the *name* of the update procedure and its *arguments*. The procedure must be *functional*: when applied to the same arguments, it must always have the same effect. In other words, there can be no state outside the arguments that affects the operation of the procedure. This means that the procedure call specified by the log entry can be reexecuted later. If the object being updated is in the same state as when the update was first done, it will end up in the same state after the update. By induction, this means that a sequence of log entries can be reexecuted, starting with the same objects, to produce the same objects produced in the original execution.

For this approach to work, two requirements must be satisfied:

- The update procedure must be a true function. That is, its result must not depend on any state outside its arguments, and it must have no side effects, except on the object in whose log it appears.
- The arguments must be values, either references to immutable objects, or immediate values such as integers or strings. An immediate value can be a large thing, like an array or even a list, but the entire value must be copied into the log entry.

Of course, most objects are not immutable since they are updated. However, a particular *version* of an object is immutable; changes made to the object change the version. A simple way to refer to an object version unambiguously is with the pair [object identifier, number of updates]. If the object identifier leads to the log for the object, replaying the specified number of log entries yields the particular version. Doing this replay may require finding some other object versions, but as long as each update refers only to existing versions, there won't be any cycles and this process will terminate.

For example, the Bravo editor[36] has exactly two update functions for editing a document:

Replace[*old*: *Interval, new*: *Interval*]
ChangeProperties[*where*: *Interval,*
what: *FormattingOp*]

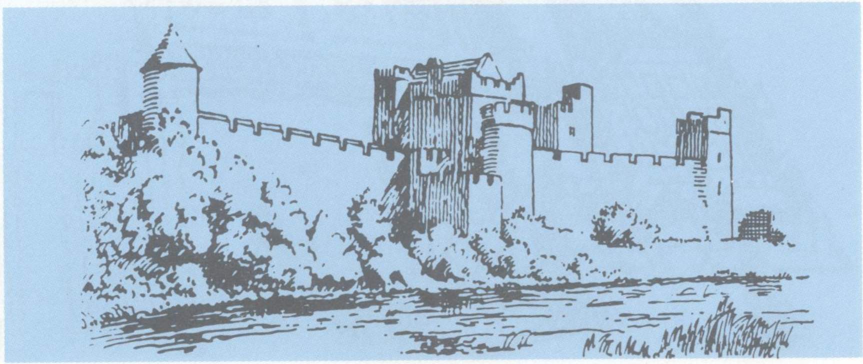
An *Interval* is a triple [document version, first character index, last character index]. A *FormattingOp* is a function from properties to properties: a property might be *italic* or *leftMargin,* and a *FormattingOp* might be *leftMargin*: = *leftMargin* + 10 or *italic*: = TRUE. Thus, only two kinds of log entries are needed. All editing commands reduce to applications of these two functions.

> **Beware**
> **Of entrance to a quarrel;**
> **but being in,**
> **Bear't that th' opposed**
> **may beware of thee.**
> —*Hamlet,* I,iii,65

Make actions atomic or restartable. An atomic action, often called a *transaction,* is one that either completes or has no effect. For example, in most main storage systems, fetching or storing a word is atomic. The advantages of atomic actions for fault tolerance are obvious. If a failure occurs during the action, it has no effect. Thus, in recovering from a failure, it is not necessary to deal with any of the intermediate states of the action.[30]

Atomicity has been provided in database systems for some time[49] by using a log to store the information needed to complete or cancel an action. The basic idea is to assign a unique identifier to each atomic action, using it to label all log entries associated with the action. A *commit record* for the action[52] tells whether it is in progress, committed (that is, logically complete, even if some cleanup work remains to be done), or aborted (that is, logically canceled, even if some cleanup remains to be done). Changes in the state of the commit record are also recorded as log entries. An action cannot be committed unless there are log entries for all of its updates. After a failure, recovery applies the log entries for each committed action and undoes the updates for each aborted action. Many variations on this scheme are possible.[53]

For this method to work, a log entry usually needs to be *restartable*. This means that any sequence of entries can be partially executed any number of times before a complete execution without changing the re-



sult. Such an action is sometimes called *idempotent.* For example, storing a set of values into a set of variables is a restartable action; incrementing a variable by one is not. Restartable log entries can be applied to the current state of the object; there is no need to recover an old state.

This basic method can be used for any kind of permanent storage. If things are simple enough, a rather distorted version will work. For example, the Alto file system described earlier in effect uses the disk labels and leader pages as a log, rebuilding its other data structures from these if necessary. Here, as in most file systems, only the file allocation and directory actions are atomic; the file system does not help the client to make its updates atomic.

The Juniper file system[50,51] goes much further, allowing each client to make an arbitrary set of updates as a single atomic action. It uses a trick known as *shadow pages,* in which data pages are moved from the log into the files simply by changing the pointers to them in the B-tree that implements the map from file addresses to disk addresses. This trick was first used in the Cal system.[27] Cooperating clients of an ordinary file system can also implement atomic actions by checking whether recovery is needed before each access to a file and, when it is, carrying out the entries in specially named log files.[54]

Atomic actions are not trivial to implement, although the preceding discussion tries to show that they are not nearly as difficult as their public image suggests. Sometimes a weaker but cheaper method will do.

The Grapevine mail transport and registration system,[55] for example, maintains a replicated database of names and distribution lists on a large number of machines in a nationwide network. Updates are made at one site and propagated to other sites using the mail system. This guarantees that the updates will eventually arrive, but as sites fail and recover and the network partitions, the order in which they arrive may vary greatly. Each update message is time-stamped, and the latest one wins. After enough time has passed, all sites will receive all updates and all will agree. During the propagation, however, the sites may disagree, for example, about whether a person is a member of a certain distribution list. Such occasional disagreements and delays are not very important to the usefulness of this particular system.

Most humbly do I take my leave, my lord.

—*Hamlet,* I,iii,82

Such a collection of good advice and anecdotes is rather tiresome to read. Perhaps it is best taken in small doses at bedtime. In extenuation, I can only plead that I have ignored most of these rules at least once, and nearly always regretted it. The references tell fuller stories about the systems or techniques I have only sketched. Many of them also have more complete rationalizations. ■

References

1. F. B. Brooks, *The Mythical Man-Month,* Addison-Wesley, Reading, Mass., 1975.
2. K. H. Britton et al., "A Procedure for Designing Abstract Interfaces for Device Interface Modules," *Proc. Fifth Int'l Conf. Software Eng.,* IEEE Computer Society Order No. 332, 1981, pp. 195-204.
3. D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Comm. ACM,* Vol. 15, No. 12, Dec. 1972, pp. 1053-1058.
4. C. A. R. Hoare, "Hints on Programming Language Design," *Proc. Sigact/Sigplan Symp. Principles of Programming Languages,* Boston, Mass., Oct. 1973.
5. M. McNeil and W. Tracz, "PL/I Program Efficiency," *Sigplan Notices,* Vol. 15, No. 6, June 1980, pp. 46-60.
6. B. W. Lampson and R. F. Sproull, "An Open Operating System for a Single-User Machine," *Operating Systems Rev.,* Vol. 13, No. 5, Dec. 1979, pp. 98-105.
7. R. R. Burton et al., "Interlisp-D Overview," in *Papers on Interlisp-D,* Technical Report SSL-80-4, Xerox Palo Alto Research Center, Palo Alto, Calif., 1981.
8. D. D. Redell et al., "Pilot: An Operating System for a Personal Computer," *Comm. ACM,* Vol. 23, No. 2, Feb. 1980, pp 81-91.
9. P. A. Janson, "Using Type-Extension to Organize Virtual-Memory Mechanisms," *Operating Systems Rev.,* Vol. 15, No. 4, Oct. 1981, pp. 6-38.
10. D. G. Bobrow et al., "Tenex: A Paged Time-Sharing System for the PDP-10," *Comm. ACM,* Vol. 15, No. 3, Mar. 1972, pp. 135-143.
11. D. Ingalls, "The Smalltalk Graphics Kernel," *Byte,* Vol. 6, No. 8, Aug. 1981, pp. 168-194.
12. W. M. Newman and R. F. Sproull, *Principles of Interactive Computer Graphics,* McGraw-Hill, New York, 2nd ed., 1979.
13. D. W. Clark et al., "The Memory System of a High-Performance Personal Computer," *IEEE Trans. Computers,* Vol. TC-30, No. 10, Oct. 1981, pp. 715-733.
14. D. E. Knuth, "An Empirical Study of Fortran Programs," *Software—Practice & Experience,* Vol. 1, No. 2, Mar. 1971, pp. 105-133.
15. R. Sweet and J. Sandman, "Static Analysis of the Mesa Instruction Set," *Sigplan Notices,* Vol. 17, No. 4, Apr. 1982, pp. 158-166.
16. A. Tanenbaum, "Implications of Structured Programming for Machine Architecture," *Comm. ACM,* Vol. 21, No. 3, Mar. 1978, pp. 237-246.
17. G. H. Radin, "The 801 Minicomputer," *Sigplan Notices,* Vol. 17, No. 4, Apr. 1982, pp. 39-47.
18. D. A. Patterson and C. H. Sequin, "RISC I: A Reduced Instruction Set VLSI Computer," *Proc. Eighth Ann. Symp. Computer Architecture,* IEEE Computer Society Order No. 346, 1981, pp. 443-457.
19. P. M. Hansen et al., "A Performance Evaluation of the Intel iAPX 432," *Computer Architecture News,* Vol. 10, No. 4, June 1982, pp. 17-26.
20. C. P. Thacker et al., "Alto: A Personal Computer," in *Computer Structures: Principles and Examples,* Siewiorek, Bell, and Newell,

eds., McGraw-Hill, New York, 2nd ed., 1982, pp. 549-580.

21. L. P. Deutsch and C. A. Grant, "A Flexible Measurement Tool for Software Systems," *Proc. IFIP Congress 71,* North-Holland, 1971.

22. B. W. Lampson and H. E. Sturgis, "Reflections on an Operating System Design," *Comm. ACM,* Vol. 19, No. 5, May 1976, pp. 251-265.

23. C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *Comm. ACM,* Vol. 17, No. 10, Oct. 1974, pp. 549-557.

24. B. W. Lampson and D. D. Redell, "Experience with Processes and Monitors in Mesa," *Comm. ACM,* Vol. 23, No. 2, Feb. 1980, pp. 105-117.

25. D. M. Ritchie and K. Thompson, "The Unix Time-Sharing System," *Bell System Technical J.,* Vol. 57, No. 6, July 1978, pp. 1905-1930.

26. C. M. Geschke et al., "Early Experience with Mesa," *Comm. ACM,* Vol. 20, No. 8, Aug. 1977, pp. 540-553.

27. H. E. Sturgis, *A Post-Morten for a Time-Sharing System,* Technical Report CSL-74-1, Xerox Palo Alto Research Center, Palo Alto, Calif., 1974.

28. R. J. Creasy, "The Origin of the VM/370 Time-Sharing System," *IBM J. Research and Development,* Vol. 25, No. 5, Sept. 1981, pp. 483-491.

29. B. W. Lampson et al., Electronic Image Processing System, US Patent 4,203,154, May 1980.

30. B. W. Lampson and H. E. Sturgis, "Atomic Transactions," in *Distributed Systems: An Advanced Course,* Lecture Notes in Computer Science 105, Springer Verlag, New York, 1981, pp. 246-265.

31. D. K. Gifford, "Weighted Voting for Replicated Data," *Operating Systems Rev.,* Vol. 13, No. 5, Dec. 1979, pp. 150-162.

32. B. W. Lampson, "Replicated Commit," unpublished paper circulated at a workshop on fundamental principles of distributed computing, Pala Mesa, Calif., Dec. 1980.

33. D. C. Smith et al., "Designing the Star User Interface," *Byte,* Vol. 7, No. 4, Apr. 1982, pp. 242-282.

34. L. P. Deutsch and D. G. Bobrow, "An Efficient Incremental Automatic Garbage Collector," *Comm. ACM,* Vol. 19, No. 9, Sept. 1976, pp. 522-526.

35. P. Rovner, personal communication, Dec. 1982.

36. B. W. Lampson, "Bravo Manual," in *Alto Users Handbook,* Xerox Palo Alto Research Center, Palo Alto, Calif., 1976.

37. J. L. Bentley, *Writing Efficient Programs,* Prentice-Hall, Englewood Cliffs, N.J., 1982.

38. D. R. Ditzel and H. R. McLellan, "Register Allocation for Free: The C Machine Stack Cache," *Sigplan Notices,* Vol. 17, No. 4, Apr. 1982, pp. 48-56.

39. J. G. Mitchell, *Design and Construction of Flexible and Efficient Interactive Programming Systems,* Garland, New York, 1979.

40. L. P. Deutsch, "Efficient Implementation of the Smalltalk-80 System," *Proc. 11th ACM Symp. Principles of Programming Languages,* 1984.

41. J. M. McQuillan and D. C. Walden, "The Arpa Network Design Decisions," *Computer Networks,* Vol. 1, Aug. 1977, pp. 243-289.

42. R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," *Comm. ACM,* Vol. 19, No. 7, July 1976, pp. 395-404.

43. J. E. Smith, "A Study of Branch Prediction Strategies," *Proc. Eighth Ann. Symp. Computer Architecture,* IEEE Computer Society Order No. 346, 1981, pp. 135-148.

44. E. W. Dijkstra et al., "On-the-fly Garbage Collection: An Exercise in Cooperation," *Comm. ACM,* Vol. 21, No. 11, Nov. 1978, pp. 966-975.

45. L. M. Breed and R. H. Lathwell, "The Implementation of APL/360," in *Interactive Systems for Experimental Applied Mathematics,* Klerer and Reinfelds, eds., Academic Press, New York, 1968, pp. 390-399.

46. D. R. Boggs et al., "Pup: An Internetwork Architecture," *IEEE Trans. Communications,* Vol. COM-28, No. 4, Apr. 1980, pp. 612-624.

47. J. H. Saltzer et al., "End-to-End Arguments in System Design," *Proc. Second Int'l Conf. Distributed Computing Systems,* IEEE Comput- Order No. 344, 1981, pp. 509-512.

48. R. M. Needham, personal communication, Dec. 1980.

49. J. Gray et al., "The Recovery Manager of the System R Database Manager," *Computing Surveys,* Vol. 13, No. 2, June 1981, pp. 223-242.

50. J. G. Mitchell and J. A. Dion, "Comparison of Two Network-Based File Servers," *Comm. ACM,* Vol. 25, No. 4, Apr. 1982, pp. 233-245.

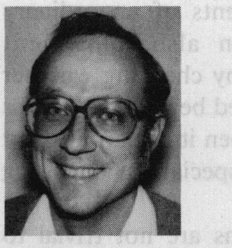
51. H. E. Sturgis et al., "Issues in the Design and Use of a Distributed File System," *Operating Systems Rev.,* Vol. 13, No. 3, July 1980, pp. 55-69.

52. D. Reed, *Naming and Synchronization in a Decentralized Computer System,* MIT LCS TR-205, MIT, Cambridge, Mass., 1978.

53. I. L. Traiger, "Virtual Memory Management for Data Base Systems," *Operating Systems Rev.,* Vol. 16, No. 4, Oct. 1982, pp. 26-48.

54. W. H. Paxton, "A Client-Based Transaction System to Maintain Data Integrity," *Operating Systems Rev.,* Vol. 13, No. 5, Dec. 1979, pp. 18-23.

55. A. D. Birrell et al., "Grapevine: An Exercise in Distributed Computing," *Comm. ACM,* Vol. 25, No. 4, Apr. 1982, pp. 260-273.



Butler W. Lampson recently joined the Systems Research Center of Digital Equipment Corporation. Earlier, he was an associate professor of computer science at the University of California, Berkeley, a founder of the Berkeley Computer Corporation, and a senior research fellow in the Computer Science Laboratory of the Xerox Palo Alto Research Center.

His address is Systems Research Center, Digital Equipment Corporation, 130 Lytton St., Palo Alto, CA 94301.

Illustrations within the article are from F. R. Cowell, *Garden as a Fine Art,* Houghton Mifflin, Boston, 1968 (pp. 12, 15, and 24); Culver Pictures, Inc., New York (p. 16); and Harold G. Leask, *Irish Castles and Castellated Houses,* Dundalgan Press, Dundalk, Ireland, 1964 (pp. 20, 23, and 26).