Hints for Computer System Design[1]

Butler W. Lampson

Computer Science Laboratory
Xerox Palo Alto Research Center
Palo Alto, CA 94304

Abstract

Studying the design and implementation of a number of computer has led to some general hints for system design. They are described here and illustrated by many examples, ranging from hardware such as the Alto and the Dorado to application programs such as Bravo and Star.

1. Introduction

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, and hence many internal interfaces.

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 any major part of it; much more important is to avoid choosing a terrible way, and to have clear division of responsibilities among the parts.

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 this experience come some general hints for designing successful systems. I claim no originality for them; most are part of the folk wisdom of experienced designers. Nonetheless, even the expert often forgets, and after the second system [6] comes the fourth one.

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.

---

[1] This paper was originally presented at the. 9th ACM Symposium on Operating Systems Principles and appeared in *Operating Systems Review* **15**, 5, Oct. 1983, p 33-48. The present version is slightly revised.

They are just hints. Some are quite general and vague; others are specific techniques which are more widely applicable than many people know. Both the hints and 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 that have already been 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, mostly drawn from systems I have worked on. They range from hardware such as the Ethernet local area network and the Alto and Dorado personal computers, through operating systems such as the SDS 940 and the Alto operating system and programming systems such as Lisp and Mesa, to application programs such as the Bravo editor and the 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 in favor of others which show unexpected uses for some well-known methods. There are references for nearly all the specific examples but for only a few of the ideas; many of these are part of the folklore, and it would take a lot of work to track down their multiple sources.

> *And these few precepts in thy memory*
> *Look thou character.*

It seemed appropriate to decorate a guide to the doubtful process of system design with quotations from *Hamlet.* Unless otherwise indicated, they are taken from Polonius' advice to Laertes (I iii 58-82). Some quotations are from other sources, as noted. Each one is intended to apply to the text which follows it.

Each hint is summarized by a slogan that when properly interpreted reveals the essence of the hint. Figure 1 organizes the slogans along two axes:

Why it helps in making a good system: with functionality (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.

Fat lines connect repetitions of the same slogan, and thin lines connect related slogans.

The body of the paper is in three sections, according to the *why* headings: functionality (section 2), speed (section 3), and fault-tolerance (section 4).

2. Functionality

The most important hints, and the vaguest, have to do with obtaining the right functionality from a system, that is, with getting it to do the things you want it to do. Most of these hints depend on the notion of an *interface* that separates an *implementation* of some abstraction from the *clients* who use the abstraction. The interface between two programs consists of the set of assumptions that each programmer needs to make about the other program in order to demonstrate the correctness of his program (paraphrased from [5]). Defining interfaces is the most important part of system design. Usually it is also the most difficult, since the interface design must satisfy three conflicting requirements: an interface should be simple, it should be complete, and it should

Why?	*Functionality* Does it work?	*Speed* Is it fast enough?	*Fault-tolerance* Does it keep working?
Where?			
Completeness	Separate normal and worst case	Shed load End-to-end Safety first	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

Figure 1: Summary of the slogans

admit a sufficiently small and fast implementation. Alas, all too often the assumptions embodied in an interface turn out to be misconceptions instead. Parnas' classic paper [38] and a more recent one on device interfaces [5] offer excellent practical advice on this subject.

The main reason interfaces are difficult to design is that each interface is a small programming language: it defines a set of objects and the operations that can be used to manipulate the objects. Concrete syntax is not an issue, but every other aspect of programming language design is present. Hoare's hints on language design [19] can thus be read as a supplement to this paper.

2.1 Keep it simple

> *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)
>
> *Those friends thou hast, and their adoption tried,*
> *Grapple them unto thy soul with hoops of steel;*
> *But do not dull thy palm with entertainment*
> *Of each new-hatch'd unfledg'd comrade.*

• *Do one thing at a time, and do it well*. An interface should capture the *minimum* essentials of an abstraction. *Don't generalize*; generalizations are generally wrong.

We are faced with an insurmountable opportunity. (W. Kelley)

When an interface undertakes to do too much its implementation will probably be large, slow and complicated. An interface is a contract to deliver a certain amount of service; clients of the interface depend on the contract, which is usually documented in the interface specification. They also depend on incurring a reasonable cost (in time or other scarce resources) for using the interface; the definition of 'reasonable' is usually not documented anywhere. If there are six levels of abstraction, and each costs 50% more than is 'reasonable', the service delivered at the top will miss by more than a factor of 10.

KISS: Keep It Simple, Stupid. (Anonymous)

If in doubt, leave if out. (Anonymous)

Exterminate features. (C. Thacker)

On the other hand,

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

Thus, 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 needed by only a few clients, unless the implementer knows how to provide them without penalizing others. A better implementer, or one who comes along ten years later when the problem is better understood, might be able to deliver, but unless *the one you have* can do so, it is wise to reduce your aspirations.

For example, PL/1 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 the cases inefficiently, but even with the optimizing compilers of 15 years later, it is hard for the programmer to tell what will be fast and what will be slow [31]. A language like Pascal or C is much easier to use, because every construct has a roughly constant cost that is independent of context or arguments, and in fact most constructs have about the same cost.

Of course, these observations apply most strongly to interfaces that clients use heavily, such as virtual memory, files, display handling, or arithmetic. It is all right to sacrifice some performance for functionality in a seldom used interface such as password checking, interpreting user commands, or printing 72 point characters. (What this really means is that though the cost must still be predictable, it can be many times the minimum achievable cost.) And such cautious rules don't apply to research whose object is learning how to make better implementations. But since research may well fail, others mustn't depend on its success.

Algol 60 was not only an improvement on its predecessors,
but also on nearly all its successors. (C. Hoare)

Examples of offering too much are legion. The Alto operating system [29] has an ordinary read/write-*n*-bytes interface to files, and was extended for Interlisp-D [7] with an ordinary paging system that stores each virtual page on a dedicated disk page. Both have small implementations (about 900 lines of code for files, 500 for paging) and are fast (a page fault takes one disk access and has a constant computing cost that is a small fraction of the disk access time, and the client

can fairly easily run the disk at full speed). The Pilot system [42] which succeeded the Alto OS follows Multics and several other systems in allowing virtual pages to be mapped to file pages, thus subsuming file input/output 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). The extra functionality is bought at a high price.

This is not to say that a good implementation of this interface is impossible, merely that it is hard. This system 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 [22], but they are tricky and easily lead to greater cost and complexity in the normal case.

> *And, in this upshot, purposes mistook*
> *Fall'n on th' inventors' heads.* (V ii 387)

Another example illustrates how easily generality can lead to unexpected complexity. The Tenex system [2] 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, and any reference it makes to an unassigned virtual page is thus similarly reported to the user program.
- Large arguments to system calls, including strings, are passed by reference.
- There is a system call CONNECT to obtain 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, to prevent guessing passwords at high speed.

CONNECT is implemented by a loop of the form

```
for i := 0 to Length(directoryPassword) do
    if directoryPassword[i] ≠ passwordArgument[i] then
            Wait three seconds; return BadPassword
    end if
end loop;
connect to directory; return Success
```

The following trick finds a password of length n in $64n$ tries on the average, rather than $128^n/2$ (Tenex uses 7 bit characters in strings). Arrange the *passwordArgument* so that its first character is the last character of a page and the next page is unassigned, and try each possible character as the first. If CONNECT reports *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. Or looked at 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.

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

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 for an interface whose importance is already known from existing uses. And be sure that you know how to make it fast.

For example, the *BitBlt* or *RasterOp* interface for manipulating raster images [21, 37] was devised by Dan Ingalls after several years of experimenting with the Alto's high-resolution interactive display. Its implementation costs 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. But the performance is nearly as good as the special-purpose character-to-raster operations that preceded it, and its simplicity and generality have made it much easier to build display applications.

The Dorado memory system [8] 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 500 MBits/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 only be justified by extensive prior experience (30 years!) with this interface, and the knowledge that memory access is usually the limiting factor in performance. Even so, it seems in retrospect that the high I/O bandwidth is not worth the cost; it is used mainly for displays, and a dual-ported frame buffer would almost certainly be better.

Finally, lest this advice seem too easy to take,

• *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 this cautionary tale shows. Word processing and office information systems usually have provision for embedding named fields in the documents they handle. For example, a form letter might have 'address' and 'salutation' fields. Usually a document is represented as a sequence of characters, and a field is encoded by something like {*name*: *contents*}. Among other operations, there is a procedure *FindNamedField* that finds the field with a given name. One major commercial system for some time used a *FindNamedField* procedure that ran in time $O(n^2)$, where n is the length of the document. This remarkable result was achieved by first writing a procedure *FindIthField* to find the *i*th field (which must take time $O(n)$ 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
end loop
```

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.

2.2 Corollaries

The rule about simplicity and generalization has many interesting corollaries.

> *Costly thy habit as thy purse can buy,*
> *But not express'd in fancy; rich, not gaudy.*

• *Make it fast*, rather than general or powerful. If it's fast, the client can program the function it wants, and another client can program some other function. It is much better to have basic operations executed quickly than more powerful ones that are slower (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.

> *Had I but time (as this fell sergeant, death,*
> *Is strict in his arrest) O, I could tell you —*
> *But let it be.* (V ii 339)

For example, many studies (such as [23, 51, 52]) have shown that programs spend most of their time doing very simple things: loads, stores, tests for equality, adding one. Machines like the 801 [41] or the RISC [39] with instructions that do these simple operations quickly can run programs faster (for the same amount of hardware) than machines like the VAX with more general and powerful instructions that take longer in the 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 [18].

To find the places where time is being spent in a large system, it is necessary to have measurement tools that will pinpoint the time-consuming code. Few systems are well enough understood to be properly tuned without such tools; it is normal for 80% of the time to be spent in 20% of the code, but *a priori* analysis or intuition usually can't find the 20% with any certainty. The performance tuning of Interlisp-D sped it up by a factor of 10 using one set of effective tools [7].

• *Don't hide power*. This slogan is closely related to the last one. 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 slight loss of performance.

For example, the Alto disk hardware [53] can transfer a full cylinder at disk speed. The basic file system [29] can transfer successive file pages to client memory at full disk speed, with time for the client to do some computing on each sector; thus 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 that match a pattern. The stream level of the file system can read or write *n* bytes to or from client memory; any portions of the *n* bytes that occupy 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; this is the only price paid for the higher level of abstraction.

• *Use procedure arguments* to provide flexibility in an interface. They can be restricted or encoded in various ways if necessary for protection or portability. This technique can greatly simplify an interface, eliminating a jumble of parameters that amount to 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.

But this theme has many variations. A more interesting example is the Spy system monitoring facility in the 940 system at Berkeley [10], which allows an untrusted user program to plant patches in the code of the supervisor. A patch is coded in machine language, but the operation that installs it checks that it does no wild branches, contains no loops, is not too long, and stores only into a designated region of memory dedicated to collecting statistics. Using the Spy, the student of the system can fine-tune his measurements without any fear of breaking the system, or even perturbing its operation much.

Another unusual example that illustrates the power of this method is the FRETURN mechanism in the Cal time-sharing system for the CDC 6400 [30]. 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.

It may be better to have a specialized language, however, if it is more amenable to static analysis for optimization. This is 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 context free recognition and call client-supplied “semantic routines” to record the results of the parse. This has obvious advantages over always building a parse tree that the client must traverse to find out what happened.

The success of monitors [20, 25] as a synchronization device 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, it provides these functions itself or calls other library facilities, and pays for what it needs. The fact that monitors give no control over the scheduling of processes waiting on monitor locks or condition variables, often cited as a drawback, 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 that is unlikely to do the right thing.

The Unix system [44] 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 and achieve precisely the effect desired.

The *end-to-end* slogan discussed in section 3 is another corollary of keeping it simple.

2.3 Continuity

There is a 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 that are 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. With a language like Mesa [15] that has complete type-checking and language support for interfaces, it is much easier to change an interface without causing the system to collapse. But even if type-checking can usually detect that an assumption no longer holds, a programmer must still correct the assumption. When a system grows to more than 250K 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 that 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* if you do have to change interfaces. Here are two rather different examples to illustrate this idea. One 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 [2] and Cal [50]) 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, the IBM 360/370 systems provided emulation of the instruction sets of older machines like the 1401 and 7090. Taken a little further, this leads to virtual machines, which simulate (several copies of) a machine on the machine itself [9].

A rather different example is the world-swap debugger, which works by writing the real memory of the target system (the one being debugged) onto a secondary storage device and reading in the debugging system in its place. The debugger then provides its user with complete access to the target world, mapping each target memory address to the proper place on secondary storage. With care it is possible to swap the target back in and continue execution. This is somewhat clumsy, but 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. It is especially useful during bootstrapping. There are many variations. For instance, the debugger can run on a different machine, with a small ‘tele-debugging’ nub in the target world that can interpret *ReadWord*, *WriteWord*, *Stop* and *Go* commands arriving from the debugger over a network. Or if the target is a process in a time-sharing system, the debugger can run in a different process.

2.4 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 [6]. If there is anything new about the function of a system, the first implementation will have to be redone completely to achieve a satisfactory (that is, acceptably small, fast, and maintainable) result. It costs a lot less if you plan to have a prototype. Unfortunately, sometimes two prototypes are needed, 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 [2]. This can even work even if the previous system was too grandiose; Unix took many ideas from Multics [44].

Even when an implementation is successful, it 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.

Give thy thoughts no tongue,
Nor any unproportion'd thought his act.

• *Keep secrets* of the implementation. Secrets are assumptions about an implementation that client programs are not allowed to make (paraphrased from [5]). 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 hard to design a good interface. And there is a tension with the desire not to hide power.

An efficient program is an exercise in logical brinkmanship. (E. Dijkstra)

There is another danger in keeping secrets. One way to improve performance is to *increase* the number of assumptions that one part of a system makes about another; the 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, a membership test takes 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.

O throw away the worser part of it,
And live the purer with the other half. (III iv 157)

• *Divide and conquer*. This is a well known method for solving a hard problem: reduce it to several 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.

A good example is in the Alto's Scavenger program, which 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 [29]. 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 arises in the Dover raster printer [26, 53], which scan-converts lists of characters and rectangles into a large $m \times n$ array of bits, in which ones correspond to spots of ink on the paper and zeros 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. The printer consumes bits faster than the available disks can deliver them, so the array cannot be stored on disk. Instead, the entire array is divided into 16×4200 bit slices called bands, and the printer electronics contains two one-band buffers. 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 filling the other buffer from the next bucket. Objects that spill over the edge of one band are added to the next bucket; this is the trick that allows the problem to be subdivided.

Sometimes it is convenient to artificially limit the resource, by *quantizing* it in fixed-size units; this simplifies bookkeeping and prevents one kind of fragmentation. The classical example is the use of fixed-size pages for virtual memory, rather than variable-size segments. 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.

> *And makes us rather bear those ills we have*
> *Than fly to others that we know not of.* (III i 81)

• *Use a good idea again* instead of generalizing it. A specialized implementation of the idea may be much more effective than a general one. The discussion of caching below gives several examples of applying this general principle. Another interesting example is the notion of *replicating* data for reliability. A small amount of data can easily be replicated locally by writing it on two or more disk drives [28]. 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 [16] shows how to solve this problem by building replicated data on top of a transactional storage system, which allows an arbitrarily large update to be done as an atomic operation (see section 4). The transactional storage itself depends on the simple local replication scheme to store its log reliably. There is no circularity here, since only the *idea* is used twice, not the code. A third way to use replication in this context is to store the commit record on several machines [27].

The user interface for the Star office system [47] has a small set of operations (type text, move, copy, delete, show properties) that apply to nearly all the objects in the system: text, graphics, file folders and file drawers, record files, printers, in and out baskets, etc. The exact meaning of an operation varies with the class of object, within the limits of what the user might find 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.

2.5 Handling all the cases

Diseases desperate grown
By desperate appliance are reliev'd
or not at all. (III vii 9)

Therefore this project
Should have a back or second, that might hold,
If this should blast in proof. (IV iii 151)

• *Handle normal and worst cases separately* as a rule, because the requirements for the two are quite different:

The normal case must be fast.

The worst case must make some progress.

In most systems it is all right to schedule unfairly and give no service to some of the processes, or even 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 processes, or even the entire system. At first this sounds terrible, but one crash a week is usually a cheap price to pay for 20% better performance. Of course the system must have decent error recovery (an application of the end-to-end principle; see section 4), but that is required in any case, since there are so many other possible causes of a crash.

Caches and hints (section 3) are examples of special treatment for the normal case, but there are many others. The Interlisp-D and Cedar programming systems use a reference-counting garbage collector [11] 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. This saves a lot of reference-counting, since most pointer assignments are to local variables. There are not very many frames, so the time to scan them is small and the collector is nearly real-time. Cedar goes farther and does not keep track of which local variables contain pointers; instead, it assumes that they all do. This means that an integer that happens to contain the address of an object which is no longer referenced will keep that object from being freed. Measurements show that less than 1% of the storage is incorrectly retained [45].

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. Cedar therefore has a conventional trace-and-sweep collector as well. This 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 happens very seldom, because 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 becomes sticky, a lot of storage will

unexpectedly become permanent. An attractive solution is to have an ‘overflow count’ table, which is 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 an 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, and the overflow table is touched only in the rare case that the count swings by more than four.

There are many cases when resources are dynamically allocated and freed (for example, real memory in a paging system), and sometimes additional resources are needed temporarily to free an item (some table might have to be swapped in to find out where to write out a page). Normally there is 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, bringing it out only in a crisis. It is necessary to bound the resources needed to free one item; this 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; this may slow things down a lot but it ensures that progress will be made.

Sometimes radically different strategies are appropriate in the normal and worst cases. The Bravo editor [24] 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 some characters are deleted, for example, the piece containing the deleted characters is split into two pieces, one pointing to the first undeleted string and the other to the second. Characters inserted from the keyboard are appended to the file, and 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 there are hundreds of pieces and things start to bog down. It is then time for a cleanup, which writes a new file containing all the characters of the document in order. Now 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 so that the user doesn’t have to stop editing (though Bravo doesn’t do this).

3. Speed

This section describes hints for making systems faster, forgoing any further discussion of why this is important. Bentley’s excellent book [55] says more about some of these ideas and gives many others.

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

• *Split resources* in a fixed way if in doubt, rather than sharing them. It is usually faster to allocate dedicated resources, it is often faster to access them, 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 because it can be tricky to allocate them intelligently, and because saving and restoring them across procedure calls may negate their speed advantages. But when programs are written in the approved modern style with lots of small procedures, 16 registers are nearly always enough for all the local variables and temporaries, so that 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 [14, 39].

Input/output 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, but when it is cheap, static allocation of computing power for various purposes is 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 itself be held in the virtual memory (as it is in several systems, including Pilot [42]), but the need to avoid circularity makes this rather complicated. Instead, real memory is dedicated to this purpose. Unless the disk is ridiculously fragmented the space thus consumed is less than the space for the code to prevent circularity.

• *Use static analysis* if you can; this is a generalization of the last slogan. Static analysis discovers properties of the program that can usually be used to improve its performance. The hooker is “if you can”; when a good static analysis is not possible, don’t delude yourself with a bad one, but fall back on a dynamic scheme.

The remarks about registers above depend on the fact that the compiler can easily 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 inter-procedural analysis that may not succeed, and in any case 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 it reads the data 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, which allows 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. Many attempts have been made to analyze programs after the fact and optimize the disk transfers, but as far as I know this has 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 software that falsify the invariant.

• *Dynamic translation* from a convenient (compact, easily modified or easily displayed) representation to one that can be quickly interpreted is an important variation on the old idea of compiling. Translating a bit at a time is the idea behind separate compilation, which goes back at

least to Fortran 2. Incremental compilers do it automatically when a statement, procedure or whatever is changed. Mitchell investigated smooth motion on a continuum between the convenient and the fast representation [34]. 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 [12] uses the bytecodes 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 that on the average 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 [14] provides a rather different example. In this device instructions are fetched into an instruction cache; as they are loaded, any operand address that is relative to the local frame pointer FP is converted into an absolute address, 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 cache, so that multiple activations of the same procedure will still work.

If thou didst ever hold me in thy heart. (V ii 349)

• *Cache answers* to expensive computations, rather than doing them over. By storing 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 is faster if $f(x)$ is needed again before it gets replaced in the cache, which presumably has limited capacity. How much faster depends on how expensive it is to compute $f(x)$. A serious problem is that when f is not functional (can give different results with the same arguments), we need a way to invalidate or update a cache entry if 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 the sum of the array elements, and Δ a new value for one of them, that is, a pair [i, v]. Then $g(x, [i, v], sum)$ is $sum - x_i + v$.

A cache that is too small to hold all the ‘active’ values will thrash, and if recomputing f is expensive performance will suffer badly. Thus it is wise to choose the cache size adaptively, making it bigger when the hit rate decreases and smaller when many entries go unused for a long time.

The classic example of caching is hardware that speeds 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 the cache, disk plays the role of main storage, and the unit of transfer is the page, segment or whatever.

But nearly every non-trivial 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 but cache all the expensive results of this computation. A change must invalidate at least the cache entries that it renders invalid; if these are too hard to identify precisely, it may invalidate more entries 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 [24] has a function *DisplayLine*(*document, firstChar*) that returns the bitmap for the line of text in the displayed document that has *document*[*firstChar*] as its first character. It also returns *lastChar* and *lastCharUsed*, the numbers of the last character displayed on the line 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 in order 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 that changes characters *i* through *j* invalidates any cache entry for which [*firstChar* .. *lastCharUsed*] intersects [*i* .. *j*]. The display is recomputed by

```
loop
    (bitMap, lastChar,  ) := DisplayLine(document, firstChar); Paint(bitMap);
    firstChar := lastChar + 1
end loop
```

The call of *DisplayLine* is short-circuited by using the cache entry for [*document, firstChar*] if it exists. At the end any cache entry that has not been 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, inter-line leading, etc. In ordinary page layout all the information about the paragraph that is needed to do the layout can be represented very compactly:

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

In the usual case this can be encoded in three or four bytes. A 30 page chapter has perhaps 300 paragraphs, so about 1k bytes are required for all this data; this is less information than is required to specify the characters on a page. Since 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 is required to render one page. Layout can be done independently for each chapter.

What makes this idea work is a cache of [*paragraph, ParagraphShape*(*paragraph*)] entries. 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.

For the apparel oft proclaims the man.

• *Use hints* to speed up normal execution. A hint, like a cache entry, is the saved result of some computation. It is different in two ways: it may be wrong, and it is not necessarily reached by an associative lookup. Because 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 rather than 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 [29] and Pilot [42] 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 that contains the page and the number of that page in the file. Page zero of each file is called the 'leader page' and contains, 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.

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. Each system therefore maintains hints to speed up these operations. Both systems represent directory by a file that contains 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. The Alto uses a link in each label that points to the next label; this makes it fast to get from page n to page $n + 1$. Pilot uses 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 that keeps track of free disk pages is a hint; the truth is represented by a special value in the label of a free page, which 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 [32]. Each node in the network keeps a table that gives the best route to each other node. 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 neighbors. Because these broadcast messages are not synchronized and are not guaranteed to be delivered, 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 [33], in which lack of a carrier signal on the cable is used as 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 then try again. If n successive collisions occur, this is taken as a hint that the number of senders is 2^n, and each sender sets the mean of its random delay interval to 2^n times its initial value. This 'exponential backoff' ensures that the net does not become overloaded.

A very different application of hints speeds up execution of Smalltalk programs [12]. In Smalltalk the code executed when a procedure is called is determined dynamically by 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 of pairs [procedure name, address of code], and when this call is executed, *Print* is looked up *x*'s table (I have normalized the unusual Smalltalk terminology and syntax, and oversimplified a bit). This is expensive. It turns out that usually the type of *x* is the same as it was last time. So 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

lastT := *Pop*(); *x* := *Pop*(); *t* := type of *x*;
if $t \neq lastT$ **then** *LookupAndCall*(*x*, "Print") **else** the body of the procedure **end if**.

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. If the type is the same next time, the procedure is called directly. Measurements show that this cache hits about 96% 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 $t \neq lastT$ can be arranged so that it normally does not branch.

The same idea in a different guise is used in the S-1 [48], which has an extra bit for each instruction in its instruction cache. It clears the bit when the instruction is loaded, sets it when the instruction causes a branch to be taken, and uses it to choose the path that the instruction fetch unit follows. If the prediction turns out to be wrong, it changes the bit and follows the other path.

• *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 one that may work well if certain assumptions are satisfied. For example, Ken Thompson's chess machine Belle relies mainly on special-purpose hardware to generate moves and evaluate positions, rather than on sophisticated chess strategies. Belle has won the world computer chess championships several times. Another instructive example is the success of personal computers over time-sharing systems; the latter include much more cleverness and have many fewer wasted cycles, but the former are increasingly recognized as the most cost-effective way to do interactive computing.

Even an asymptotically faster algorithm is not necessarily better. There is an algorithm that multiplies two $n \times n$ matrices faster than $O(n^{2.5})$, but 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 can'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: first, a rapid response is better for the users, and second, the load usually varies a great deal, so there is likely to be idle processor time later in which to do background work. Many kinds of work can be deferred to background. The Interlisp and Cedar garbage collectors [7, 11] 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 the on-the-fly garbage collection algorithm given in [13]. But in most cases a simple producer-consumer relationship between two otherwise independent processes is possible.

• *Use batch processing* if possible. Doing things incrementally almost always costs more, even aside from the fact that disks and tapes work much better when accessed sequentially. Also, batch processing permits much simpler error recovery. The Bank of America has an interactive system that allows tellers to record deposits and check withdrawals. It is loaded with current account balances in the morning and does its best to maintain them during the day. But early the next morning the on-line data is discarded and replaced with the results of night's batch run. This design makes it much easier to meet the bank's requirements for trustworthy long-term data, and there is no significant loss in function.

Be wary then; best safety lies in fear. (I iii 43)

• *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 has made it clear that a general-purpose system cannot optimize the use of resources. 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 like processor cycles or communication bandwidth to be utilized more fully.

The sad truth about optimization was brought home by the first paging systems. In those days memory was very expensive, and 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 share data or code, etc. No one ever learned how to do this. Instead, memory got cheaper, and systems spent it to provide enough cushion for simple demand paging to work. We learned that the only important thing is to avoid thrashing, or too much demand for the available memory. A system that thrashes spends all its time waiting for the disk.

The only systems in which cleverness has worked are those with very well-known loads. For instance, the 360/50 APL system [4] has the same size workspace for each user and common system code for all of them. It makes all the system code resident, allocates a contiguous piece of disk for each user, and overlaps a swap-out and a swap-in with each unit of computation. This works fine.

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

A similar lesson was learned about processor time. With 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: interactive vs. non-interactive computation or high, foreground and background priority levels. The most successful schemes give a fixed share of the cycles to each job and don't allocate more than 100%; 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.

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

• *Shed load* to control demand, rather than allowing the system to become overloaded. This is a corollary of the previous rule. There are many ways to shed load. An interactive system can refuse new users, or even deny service to existing ones. A memory manager can limit the jobs being served so that all their working sets fit in the available memory. A network can discard packets. If it comes to the worst, the system can crash and start over more prudently.

Bob Morris suggested that a shared interactive system should have a large red button on each terminal. The user pushes the button if he is dissatisfied with the service, and the system must either improve the service or throw the user off; it makes 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.

The original specification for the Arpanet [32] was that a packet accepted by the net is guaranteed to be delivered unless the recipient machine is down or a network node fails while it is holding the packet. This turned out to be a bad idea. This rule makes it very hard to avoid deadlock in the worst case, and attempts to obey it lead to many complications and inefficiencies even in the normal case. Furthermore, the client does not benefit, since it still has to deal with packets lost by host or network failure (see section 4 on end-to-end). Eventually the rule was abandoned. The Pup internet [3], faced with a much more variable set of transport facilities, has always ruthlessly discarded packets at the first sign of congestion.

4. Fault-tolerance

> *The unavoidable price of reliability is simplicity.* (C. Hoare)

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.

> *This above all: to thine own self be true,*
> *And it must follow, as the night the day,*
> *Thou canst not then be false to any man.*

• *End-to-end*. Error recovery at the application level is absolutely necessary for a reliable system, and any other error detection or recovery is not logically necessary but is strictly for performance. This observation was first made by Saltzer [46] and 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. To be confident that the right bits are really on B's disk, you must 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 of the bits on 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, the bits might be clobbered while sitting in memory, or whatever. These other checks are not necessary either, since if the end-to-end check fails the entire transfer can be repeated. Of course this is a lot of work, and if errors are frequent, intermediate checks can reduce the amount of work that must be repeated. But this is strictly a question of performance, irrelevant to the reliability of the file transfer. Indeed, in the ring based system at Cambridge it is customary to copy an entire disk pack of 58 MBytes with only an end-to-end check; errors are so infrequent that the 20 minutes of work very seldom needs to be repeated [36].

Many uses of hints are applications of this idea. In the Alto file system described earlier, for example, the check of the label on a disk sector before writing the sector ensures that 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 [4] adopts the end-to-end strategy at several levels. The main service offered by the network is transport of 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 that store and forward 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 wrong 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, and 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. However, the packet transport does attempt to report problems to its clients, by providing a modest amount of error control (a 16-bit checksum), notifying senders of discarded packets when possible, etc. These services are intended to improve performance in the face of unreliable communication and overloading; since they too are best efforts, they don't complicate the implementation much.

There are two problems 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 pressures past,
That youth and observation copied there;
And thy commandment all alone shall live
Within the book and volume of my brain,
Unmix'd with baser matter. (I v 97)

• *Log updates* to record the truth about the state of an object. A log is a very simple data structure that can be reliably written and read, and cheaply forced out onto disk or other stable storage that can survive a crash. Because it is append-only, the amount of writing is minimized, and it is

fairly easy to ensure that the log is valid no matter when a crash occurs. It is also easy and cheap to duplicate the log, write copies on tape, or whatever. Logs have been used for many years to ensure that information in a data base is not lost [17], but the idea is a very general one and can be used in ordinary file systems [35, 49] 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 is no cheap way to check its correctness).

To use the technique, record every update to an object as a log entry consisting 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 is 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 re-executed later, and 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 as after the update was first done. By induction, this means that a sequence of log entries can be re-executed, starting with the same objects, and produce the same objects that were produced in the original execution.

For this to work, two requirements must be satisfied:

- The update procedure must be a true function:

  Its result does not depend on any state outside its arguments.

  It has no side effects, except on the object in whose log it appears.

- The arguments must be *values*, one of:

  Immediate values, such as integers, strings, etc. 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.

  References to *immutable* objects.

Most objects of course 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 that object, then replaying the specified number of log entries yields the particular version. Of course 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 [24] 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, last character]. 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 the 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.

• *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, so that in recovering from a failure it is not necessary to deal with any of the intermediate states of the action [28]. Database systems have provided atomicity for some time [17], 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 and use it to label all the log entries associated with that action. A commit record for the action [42] tells whether it is in progress, committed (logically complete, even if some cleanup work remains to be done), or aborted (logically canceled, even if some cleanup remains); 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 [54].

For this to work, a log entry usually needs to be restartable. This means that it can be partially executed any number of times before a complete execution, without changing the result; sometimes such an action is 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. The Alto file system described above, for example, in effect uses the disk labels and leader pages as a log and rebuilds its other data structures from these if necessary. As in most file systems, it is only the file allocation and directory actions that are atomic; the file system does not help the client to make its updates atomic. The Juniper file system [35, 49] 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 [50]. Cooperating clients of an ordinary file system can also implement atomic actions, by checking whether recovery is needed before each access to a file; when it is they carry out the entries in specially named log files [40].

Atomic actions are not trivial to implement in general, although the preceding discussion tries to show that they are not nearly as hard as their public image suggests. Sometimes a weaker but cheaper method will do. The Grapevine mail transport and registration system [1], for example, maintains a replicated data base 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 itself. 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 the sites will receive all the updates and will all 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.

5. Conclusion

Most humbly do I take my leave, my lord.

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 that I have only sketched. Many of them also have more complete rationalizations.

All the slogans are collected in Figure 1 near the beginning of the paper.

Acknowledgments

I am indebted to many sympathetic readers of earlier drafts of this paper and to the comments of the program committee.

References

1. Birrell, A.D. *et al*. Grapevine: An exercise in distributed computing. *Comm. ACM* **25**, 4, April 1982, pp 260-273.
2. Bobrow, D.G. *et al*. Tenex: A paged time-sharing system for the PDP-10. *Comm. ACM* **15**, 3, March 1972, pp 135-143.
3. Boggs, D.R. *et al*. Pup: An internetwork architecture. *IEEE Trans. Communications* **COM-28**, 4, April 1980, pp 612-624.
4. Breed, L.M and Lathwell, R.H. The implementation of APL/360. In *Interactive Systems for Experimental Applied Mathematics*, Klerer and Reinfelds, eds., Academic Press, 1968, pp 390-399.
5. Britton, K.H., *et al*. A procedure for designing abstract interfaces for device interface modules. *Proc. 5th Int'l Conf. Software Engineering*, IEEE Computer Society order no. 332, 1981, pp 195-204.
6. Brooks, F.H. *The Mythical Man-Month*, Addison-Wesley, 1975.
7. Burton, R.R. *et al*. Interlisp-D overview. In *Papers on Interlisp-D*, Technical report SSL-80-4, Xerox Palo Alto Research Center, 1981.
8. Clark, D.W. *et al*. The memory system of a high-performance personal computer. *IEEE Trans. Computers* **TC-30**, 10, Oct. 1981, pp 715-733.
9. Creasy, R.J. The origin of the VM/370 time-sharing system. *IBM J. Res. Develop.* **25**, 5, Sep. 1981, pp 483-491.
10. Deutsch, L.P. and Grant. C.A. A flexible measurement tool for software systems. *Proc. IFIP Congress* 1971, North-Holland.
11. Deutsch, L.P. and Bobrow, D.G. An efficient incremental automatic garbage collector. *Comm. ACM* **19**, 9, Sep. 1976, pp 522-526.

12. Deutsch, L.P. Efficient implementation of the Smalltalk-80 system. *Proc. 11th ACM Symposium on Principles of Programming Languages*, 1984..

13. Dijkstra. E.W. *et al*. On-the-fly garbage collection: An exercise in cooperation. *Comm. ACM* **21**, 11, Nov. 1978, pp 966-975.

14. Ditzel, D.R. and McClellan, H.R. Register allocation for free: The C machine stack cache. *SIGPLAN Notices* **17**, 4, April 1982, pp 48-56.

15. Geschke, C.M, *et al*. Early experience with Mesa. *Comm. ACM* **20**, 8, Aug. 1977, pp 540-553.

16. Gifford, D.K. Weighted voting for replicated data. *Operating Systems Review* **13**, 5, Dec. 1979, pp 150-162.

17. Gray, J. *et al*. The recovery manager of the System R database manager. *Computing Surveys* **13**, 2, June 1981, pp 223-242.

18. Hansen, P.M. *et al*. A performance evaluation of the Intel iAPX 432, *Computer Architecture News* **10**, 4, June 1982, pp 17-26.

19. Hoare, C.A.R. Hints on programming language design. *SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, Boston, Oct. 1973.

20. Hoare, C.A.R. Monitors: An operating system structuring concept. *Comm. ACM* **17**, 10, Oct. 1974, pp 549-557.

21. Ingalls, D. The Smalltalk graphics kernel. *Byte* **6**, 8, Aug. 1981, pp 168-194.

22. Janson, P.A. Using type-extension to organize virtual-memory mechanisms. *Operating Systems Review* **15**, 4, Oct. 1981, pp 6-38.

23. Knuth, D.E. An empirical study of Fortran programs, *Software–Practice and Experience* **1**, 2, Mar. 1971, pp 105-133.

24. Lampson. B.W. Bravo manual. In *Alto Users Handbook*, Xerox Palo Alto Research Center, 1976.

25. Lampson, B.W. and Redell, D.D. Experience with processes and monitors in Mesa. *Comm. ACM* **23**, 2, Feb. 1980, pp 105-117.

26. Lampson, B.W. *et al*. Electronic image processing system, U.S. Patent 4,203,154, May 1980.

27. Lampson, B.W. Replicated commit. Circulated at a workshop on Fundamental Principles of Distributed Computing, Pala Mesa, CA, Dec. 1980.

28. Lampson, B.W. and Sturgis, H.E. Atomic transactions. In *Distributed Systems — An Advanced Course*, Lecture Notes in Computer Science **105**, Springer, 1981, pp 246-265.

29. Lampson, B.W. and Sproull, R.S. An open operating system for a single-user machine. *Operating Systems Review* **13**, 5, Dec. 1979, pp 98-105.

30. Lampson, B.W. and Sturgis, H.E. Reflections on an operating system design. *Comm. ACM* **19**, 5, May 1976, pp 251-265.

31. McNeil, M. and Tracz, W. PL/1 program efficiency. *SIGPLAN Notices* **5**, 6, June 1980, pp 46-60.

32. McQuillan, J.M. and Walden, D.C. The ARPA network design decisions. *Computer Networks* **1**, Aug. 1977, pp 243-299.

33. Metcalfe, R.M. and Boggs, D.R. Ethernet: Distributed packet switching for local computer networks. *Comm. ACM* **19**, 7, July 1976, pp 395-404.

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

35. Mitchell, J.G. and Dion, J. A comparison of two network-based file servers. *Comm. ACM* **25**, 4, April 1982, pp 233-245.

36. Needham, R.M. Personal communication. Dec. 1980.

37. Newman, W.M. and Sproull, R.F. *Principles of Interactive Computer Graphics*, 2nd ed., McGraw-Hill, 1979.

38. Parnas, D.L. On the criteria to be used in decomposing systems into modules. *Comm. ACM* **15**, 12, Dec. 1972, pp 1053-1058.

39. Patterson, D.A. and Sequin, C.H. RISC 1: A reduced instruction set VLSI computer. *8th Symp. Computer Architecture*, IEEE Computer Society order no. 346, May 1981, pp 443-457.

40. Paxton, W.H. A client-based transaction system to maintain data integrity. *Operating Systems Review* **13**, 5, Dec. 1979, pp 18-23.

41. Radin, G.H. The 801 minicomputer, *SIGPLAN Notices* **17**, 4, April 1992, pp 39-47.

42. Redell, D.D. *et al*. Pilot: An operating system for a personal computer. *Comm. ACM* **23**, 2, Feb. 1980, pp 81-91.

43. Reed, D. *Naming and Synchronization in a Decentralized Computer System*, MIT LCS TR-205. Sep. 1978.

44. Ritchie, D.M. and Thompson, K. The Unix time-sharing system. *Bell System Tech. J.* **57**, 6, July 1978, pp 1905-1930.

45. Rovner, P. Personal communication. Dec. 1982.

46. Saltzer, J.H. *et al.* End-to-end arguments in system design. *Proc. 2nd Int'l. Conf. Distributed Computing Systems*, Paris, April 1981, pp 509-512.

47. Smith, D.C. *et al*. Designing the Star user interface. *Byte* **7**,4, April 1982, pp 242-282 .

48. Smith, J.E. A study of branch prediction strategies. *8th Symp. Computer Architecture*, IEEE Computer Society order no. 346, May 1981, pp 135-148.

49. Sturgis, H.E, *et al*. Issues in the design and use of a distributed file system. *Operating Systems Review* **14**, 3, July 1980, pp 55-69.

50. Sturgis, H.E. *A Postmortem for a Time Sharing System*. Technical Report CSL-74-l, Xerox Palo Alto Research Center, 1974.

51. Sweet, R., and Sandman, J. Static analysis of the Mesa instruction set. *SIGPLAN Notices* **17**, 4, April 1982, pp 158-166.

52. Tanenbaum, A. Implications of structured programming for machine architecture. *Comm. ACM* **21**, 3, March 1978, pp 237-246.

53. Thacker, C.P. *et al*. Alto: A personal computer. In *Computer Structures: Principles and Examples*, 2nd ed., Siewiorek, Bell, and Newell, eds., McGraw-Hill,1982.

54. Traiger, I.L. Virtual memory management for data base systems. *Operating Systems Review* **16**, 4, Oct. 1982, pp 26-48.

55. Bentley, J.L. *Writing Efficient Programs*. Prentice-Hall, 1982.