QUAESTIONES INFORMATICAE

Vol. 1 No. 3

March, 1980

Quaestiones Informaticae

An official publication of the Computer Society of South Africa
'n Amptelike tydskrif van die Rekenaarvereniging van Suid-Afrika

Editors: Dr. D. S. Henderson,
Vice Chancellor, Rhodes University, Grahamstown, 6140, South africa.
Prof. M. H. Williams,
Department of Computer Science and Applied Maths,
Rhodes University, Grahamstown, 6140, South Africa.

Subscriptions

Annual subscriptions are as follows:

	SA	US	UK
Individuals	R2	$3	£1.50
Institutions	R4	$6	£3.00

Quaestiones Informaticae is prepared for publication by SYSTEMS PUBLISHERS (PTY) LTD

for the Computer Society of South Africa.

The Memory Organization of Future Large Processors

David M. Stein

IBM T.J. Watson Research Center, Yorktown Heights, New York 10598

Abstract

In this paper some specifics and some generalities on the subject of memory organization for future large processors are discussed. We briefly review the current hierarchy and discuss technological and economic changes which are expected to occur in the near future. Using simple models, we show how the impact of these changes will be felt and explain that the memory hierarchy will become an increasingly important part of future computers. We show that a need is developing for understanding the memory organization in a "Systems" sense, and we draw upon theories of organization and hierarchies to elaborate upon the issues and to gain insight.

1. Introduction

In this paper we discuss some specifics and some generalities on the subject of memory organization for future large processors. Most large computers are at present centralized in that the central processing unit (CPU) controls the whole machine. In these systems the primary role of the memory hierarchy is to stage data to the CPU.

We wish to place the memory into a functional perspective. To do so, we begin by briefly reviewing the current hierarchy and discussing technological and economic changes which are expected to occur in the near future. These developments will of course affect the memory hierarchy. In Section 2 we show how the impact of new constraints and faster CPU's will be felt. Simple models are used to highlight new problems which will arise and to explain the fact that the memory hierarchy will become an increasingly important part of future computers. It is shown that a need is developing to understand the memory organization in a "Systems" sense. We discuss these aspects of the memory in Section 3 and we draw upon theories of organization and hierarchies to elaborate upon the issues and to gain insight. An important issue is the decentralization of the intelligence of the CPU.

1.1. Current Memory Hierarchies

The main role of the memory is to provide the CPU with instructions and data. For economic reasons the memory is implemented as levels in an hierarchy: the aim is to provide a memory which will appear to the CPU as if it had the speed of its fastest level with the size of its slowest. In Figure 1 we show the hierarchy levels and their approximate sizes and speeds in current large machines (the numbers shown are for machines such as the IBM 3033). In the CPU itself there are registers; next is the cache, which is transparent to the software — cache "lines" are brought from the next level, main memory. Then come the disks (we include among these other direct access storage devices, such as drums) from where the main memory receives its "pages". This takes several orders of magnitude longer than the few machine cycles penalty for a cache line miss, so while the page is being fetched the CPU begins executing another task. The number of tasks needed to keep the CPU busy is the multiprogramming level (MPL). Finally there is a mass storage medium (e.g. magnetic tape or other). In this paper we do not explicitly consider the CPU registers, which are really part of the CPU, or the final level, the mass store.

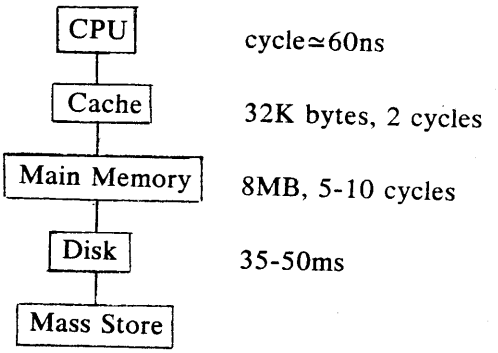


FIGURE 1

The hierarchical implementation of memory works because of a property of locality that exhibits itself in many forms in program behaviour. Thus, when a required memory reference is not found in one level of the hierarchy, a larger block of code (a line, page, etc.) is staged into that level from higher levels.

Numerous design problems are associated with the memory hierarchy. These are both of an economic nature — how large and how fast each level should be — and of an algorithmic nature — when to stage in data, how to store it, and when to evict it. Some recent references are [4,8]. We do not wish to discuss these design issues in detail here, but rather to focus upon new trends and how they will affect the memory hierarchy.

Note that the implementation shown in Fig. 1 is largely "centralized" in that the CPU provides the mechanisms whereby data is staged. We shall refer to this as the "classical" hierarchy.

1.2 Technological and Economic Trends

Computer technologies are currently advancing at a tremendous rate. These technologies promise many benefits such as faster processors, faster memories and new memory devices. At the same time both the engineering constraints and the economics are changing.

We outline here some technological and economic trends that appear to affect memory designs most strongly. The reader is referred to [1,6] for details of these developments.

With VLSI, greatly improved speeds will be possible. Improvements will be seen both in the speeds of logic and in the speeds of

semiconductor memories. In addition there will be substantial reductions in cost. Logical functions will become much cheaper to implement, and improved CPU designs will be possible. On the other hand new constraints are appearing. Costs will be determined by the number of chips and not by the number of circuits. There will be limitations on the number of pins per chip and communication delays between chips will increase. New problems will be created by packaging and heating constraints.

Changes will also be seen in the memory devices themselves. New memory technologies are already appearing — examples are CCD's, Bubbles, EBAM and many others. Improved speeds are possible and again there are new economic criteria. An immediate question is that of where in the memory hierarchy these new devices might best be used.

In some areas advances are relatively slow. Access times to electromechanical disks are not decreasing since large improvements in rotational speeds do not appear to be possible. To reduce the cost/bit rates of storage on disks, higher track densities are appearing.

In the following sections we shall discuss the implications of these trends for the memory organization. It will be seen that the importance of the processing resource is diminishing, while problems of communication are increasing. Wire and path delays and the importance of physical distance will imply the growing need to optimize the use of the memory resource.

2. Performance of the Classical Memory Hierarchy

In discussing the memory organization of future processors we focus mainly upon performance. In this section we discuss the classical hierarchy, shown in Fig. 1. This is a centralized implementation, and its performance can be measured in terms of effects upon the CPU. We thus begin with the performance of the CPU and develop outwards to discuss the main memory and other system effects. Our aim is to investigate the effects of the technological changes which are appearing. Our method is to use very simple models: although major assumptions are made, the models focus upon important parameters and allow us easily to grasp the most important principles.

2.1 Performance of the CPU

One measure of the speed of the CPU is the well known quantity "MIPS" — millions of instructions per second. This notion has some important limitations; however, MIPS is an easily understood quantity and permits a simplicity of exposition. So we retain the essential concept, yet adapt it to make a few points.

The speed at which a CPU can be driven depends upon the organization of the memory which is attached to it. To study this relationship, let us define the raw speed of the CPU — which is independent of the memory — as:

IMIPS = Infinite cache MIPS.

This MIPS rate would be obtained in the ideal case in which the processor had an infinite cache. There would then never be a need to access main memory, and no access penalties would be incurred. IMIPS can be expressed in terms of some fundamental parameters. Let γ be the cycle time of the CPU in nanoseconds, k be the average rate of cycles/instruction (with an infinite cache).

Then:

$$\text{IMIPS} = 1000/\gamma k$$

IMIPS can be increased by reducing either γ or k. Improvements in γ are obtained when faster technologies are used. If, for example, a current machine were directly mapped into a faster technology, so that all switching and logic delays were made proportionally smaller, then γ would decrease. A reduction in k can be obtained by CPU design improvements. For example, by taking advantage of cheaper logic, pipelining or concurrency can be increased, and the instruction and execution units can be streamlined to yield lower cycle/instruction rates.

2.2. Performance of Main Memory

In reality the cache is finite and a main memory supports it. A notion of "finite cache MIPS" — FMIPS — shows how the raw performance of the CPU is affected by main memory.

Let there be, on average, r memory references per instruction, m cache misses per reference, p cycles penalty for each cache miss. A penalty is incurred by the CPU for each fetch reference that misses from the cache: the CPU loses some of its processing power. For simplicity one can imagine the CPU as stopping when a fetch reference misses, and remaining idle until the line appears in the cache. In this case, p is a sum of path delays, of queuing delays for the required busses and the memory module, and of a memory access time. In actuality, the delay p also depends upon prefetching mechanisms and upon the pipelined organization of the CPU.

Using the parameters above, the average penalty due to cache misses is rmp cycles per instruction. Hence, the new cycle/instruction rate is k+rmp, and we can define the finite-cache MIPS as:

$$\text{FMIPS} = 1000/\gamma(k+rmp) \qquad (1)$$

To fix ideas, typical values for the parameters might be:

$$\begin{aligned} \gamma &= 60\text{ns} \\ k &= 3.2 \text{ cycles/instruction} \\ r &= 1.8 \text{ references/instruction} \\ m &= .03 \\ p &= 10 \text{ cycles} \end{aligned} \qquad (2)$$

Note that our notion of FMIPS corresponds to the usual term MIPS. Having expressed IMIPS and FMIPS in terms of basic parameters we can investigate some future implications for the memory hierarchy. What happens, for example, when γ and k are reduced by technological and design improvements?

Consider the cache miss ratio, m. Current processors have large caches and increasing the cache size would yield only minor improvements in m. With improved CPU designs we might expect the reference rate r to grow; for example, by prefetching additional instructions down unresolved branches, operations might be streamlined. Possibly even more important, however, is the fact that the path delay included in p will remain high: limitations on the number of pins will restrict bandwidth, and communication delays incurred when signals are sent off chip and across boards will be large.

For simplicity, let us assume first that the product rmp will remain at its current value, and consider the fraction of *usable* IMIPS;

$$\text{FMIPS/IMIPS} = k/(k+rmp) \qquad (3)$$

This fraction is independent of γ, the cycle time: changes in technology *only* should not affect the fraction of usable IMIPS. But, by improving the CPU design (a reduction in k) this fraction will decrease. For the numbers shown previously, rmp =.54; only .85 of the IMIPS are actually used. When k = 1.5 it is .73, and when k = 1 the fraction of usable IMIPS is only .65. Thus, improvements in k will have decreasing returns.

Expressed another way, by differentiating (3), it is seen that the rate of decrease of FMIPS/IMIPS with respect to k is inversely proportional to the *square* of the IMIPS.

In addition, note that from (1) we have that FMIPS becomes more sensitive to the access time of main memory as k decreases.

Because of this sensitivity to the access time, an extremely fast memory will be needed. For example, if the CPU cycle is 15ns and the access time is 5 cycles, a memory access of 75ns is required. With k = 1.5, if the memory access is now increased to 90ns, there will be a loss in FMIPS of over 3%. Despite the fact that memory costs are decreasing, memories of this speed will still be expensive, and the cost of main memory will still be high.

We have discussed here only a simple design in which the CPU is idle during cache faults, and we have motivated an argument for rmp to remain constant. It may indeed be possible to reduce rmp. For example, one might prefetch cache lines in early anticipated use, but this could potentially create bottlenecks on the bus or force useful information from the cache. There are complex interrelationships among even the simple parameters we have introduced, and unravelling them remains an interesting research project.

2.3. System Effects

The development of FMIPS in the last section tacitly assumed that main memory was infinite. When a finite main memory is considered, performance is a more complex issue. In order for the memory to function effectively, additional overheads are incurred. In this section we discuss these overheads and their effect upon the system; the effects are measured primarily in terms of throughput (transactions executed per second) and multiprogramming level. It is necessary too to study disks and the contention for them, and to introduce another gross measure of performance — transaction response time.

(i) Transactions and throughput

Let us imagine an "average" transaction in a multiprogrammed timesharing environment. This average transaction is comprised of T instructions. In order to execute them, a data set of P pages must be paged in from main memory. At the completion of the transaction these P pages have to be paged out again. During the execution of the transaction, additional pages are referenced — we assume that there are M such pages read or written to or from backing store.

Software overheads are incurred in the paging process: additional instruction paths are required for example to allocate memory, to set up the disk or I/O devices, to search for the records, to schedule the transactions, etc. Let us assume that there are S instructions needed initially to swap the P pages in and out of memory. At each of the M page faults we assume Q instructions overhead — this includes also a wastage due to extra cache faults since the contents of the cache become obsolete.

Typical values for these parameters might be:

$$\begin{aligned} &T = 1 \text{ million instructions} \\ &P = 20 \text{ pages, 4K bytes each} \\ &M = 50 \text{ page faults — reads or writes} \\ &S = 100\,000 \text{ instructions} \\ &Q = 6000 \text{ instructions} \end{aligned} \tag{4}$$

For each transaction, T+S+MQ instructions are executed. Of these, only T are devoted to real, constructive work. We can refer to:

$$\text{TMIPS} = (T/T+S+MQ)\text{FMIPS}$$

as "throughput MIPS". The user, requiring T instructions to be executed, sees only these TMIPS. Using (4), this is only .7 of the FMIPS.

One can now see how the memory hierarchy causes a loss in CPU performance. Using FMIPS/TMIPS = .7, less than 50% of the CPU performance is devoted to constructive work.

The relationship between TMIPS and throughput is simple; if the CPU is running at a utilization of u, then its throughput is:

$$u \times \text{FMIPS} \times 10^6/(T+S+MQ) \text{ trans/sec.}$$

(ii) Disks and contention

The performance of the disks and their effect upon the system is more complex than is indicated by TMIPS. The disks and the data paths to them are limited resources that must be allocated to the page-fault requests. One can think of the resources as being queuing servers; the requests queue up, awaiting their turn for service [9].

If we consider a disk arm as a single queue under the usual simplifying assumptions [3], the average "reaction time", R, of the disk — the time from the page fault request until the disk has responded with the required information and the task is available for continued service at the CPU — is given by:

$$R = L/(1-\rho),$$

where ρ is the utilization of the disk arm and L is its average busy time for each request. That the reaction time is important will be seen soon: it directly affects the MPL and it is the major component of transaction response time.

Contention for the disk arm appears in the fact that the server becomes highly utilized. As ρ increases to 1, queuing delays become large and R grows without bound. Typically, ρ is kept to below 30% for all arms; R is usually in the range 30-50ms.

Returning, now, to our storage system, what directions will future performance take? The actual disk service time, L, is dominated by the mechanical rotation speed of the device, and, as we have remarked, has limited potential for improvement. Most current developments in disk technology are in the area of storage density (bits per unit area). However, increasing the storage density — with fixed total capacity — merely increases the request rate to each disk arm and hence its utilization.

As FMIPS increases, the total disk request rate will increase by approximately the same factor. Since service times are not increasing, at least the same increase in the number of disk arms will be needed. Of course, the bandwidth to the disk system, being another potential bottleneck, will also have to be increased.

It can be remarked finally that data bases are growing at a very rapid rate, and very large numbers of disks will be required in the future. Since the processors themselves are becoming cheaper, a very large portion of total hardware cost will be devoted to the disk system.

(iii) The multiprogramming level

Having discussed the disks and their reaction time, we return to study their effect upon system performance. To first order, the average time between disk requests is:

$$(T+S+MQ)/u \times M \times \text{FMIPS} \times 1\,000 \text{ milliseconds}$$

The multiprogramming level, MPL, will be approximately:

$$\text{MPL} = R \times \text{FMIPS} \times M \times 1000/(T+S+MQ) \times u \tag{5}$$

Taking the nominal parameters in (3) and u = .8, we have that MPL=1.6×FMIPS. In point of fact, this argument is gross and the approximation (5) underestimates the MPL. It holds reasonably well for $u \leq .8$. But, when u is higher, queueing effects play an important role. A more accurate feel for system effects is obtained from the well known central server queueing model [5]. Before doing this we make some easy observations.

For fixed transaction parameters, (5) shows that the product R×FMIPS approximately determines the MPL. Since R will reduce only very slightly in future systems, an increase in FMIPS

will increase the MPL by the same factor. What effect will this have?

First note that in order to avoid increasing the high page-swap overheads, the main memory must hold the currently active pages of all transactions that have been admitted to the MPL. One rule of thumb might be:

Size of Main Memory=a+b×MPL,

where a and b are constants. For example, if a = 2MB and b = .5MB, with MPL = 8, a main memory size of 6MB is required. But, if MPL = 100, a memory of over 50MB is needed — an expensive proposition for sure.

Second, we have been assuming that the software overheads, S and Q, are constant. Their values, however, depend upon the MPL and increase with the scheduling overheads, the management of memory, the searching of queues, etc. Typically, the software effort required to perform these tasks well grows very fast with the size of the problems. While this would tend to stabilize the MPL in (5), there would be a large wastage of TMIPS, and a decline in throughput.

The next section indicates an approach for reducing MPL in future systems.

2.4. A Bulk Memory

The growing need for a large main memory, high MPL and a large number of disk arms has arisen because of the widening gap between the speeds of the main memory and disk. These requirements can be avoided if a new level of memory — which we shall refer to as the "bulk memory" — is inserted in the storage hierarchy between the main memory and disk.

Logically, there is no difficulty with an additional level in the storage hierarchy: requests that miss from main memory will, with a probability δ, say, be found in the bulk store; with probability $(1-\delta)$ a block of data will have to be staged in from disk. A number of technologies are available for the bulk store eg. CCDs, bubbles or EBAM; they have access times for a 4K page lying in the range of 200μs to 1ms. With regard implementation, numerous options are available, and some brief comments are made below. First, we motivate the need for a bulk memory from a performance viewpoint.

We have used a central server model [5] with parameters as described in the preceding sections to study performance in greater detail. The model provides the queueing delays, utilization of the devices, throughput etc. for various MPLs. It also provides the response time of the average transaction — this is comprised of its execution time at the CPU, the sum of its disk reaction times and its queueing delays for CPU service. In Figure 2 the relationship between response time and throughput has been plotted for a CPU with FMIPS = 50 and a bulk memory with access time of 800μs for a 4K page. Shown are the cases with a miss ratio of $\delta = \frac{1}{4}$ and $\frac{1}{8}$ — a 16K block is assumed to be transferred between disk and bulk store — and the case with no bulk store. In each case the MPL is shown at various throughput values; as the MPL and throughput increase, the CPU becomes more utilized until no higher throughput can be achieved. Included here is a submodel of channels and disk arms. If the number of disk arms required to support the 50 FMIPS system with no bulk memory is 1X, then the number of disk arms required to support the system at $\delta = .125$ and $\delta = .25$ is .3X and .4X respectively.

Fig. 2 shows clearly that a bulk memory provides performance in two ways. First, fewer accesses are made to disk, thus reducing arm contention. Second, the *average* reaction time R for a miss from main memory is reduced. This lowers the MPL and main memory requirements. At the same time, the response time of the average transaction improves.

Note that, even in the case when a bulk memory is provided, response time is mainly determined by the disk reaction time: the bulk memory access contributes only a small fraction. Hence the MPL too is mainly determined by disk reaction time. Thus the miss ratio δ crucially affects system performance. The bulk memory access time is less important; this implies that a large, though slow,

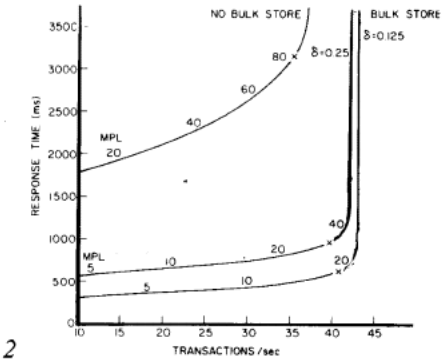


FIGURE 2

bulk memory would be preferable to a smaller, though faster one. Numerous options are available for the implementation of a bulk store. For example, it could be viewed architecturally as an extension to main memory, as a large drum replacement, or it could be a transparent buffer to disk, residing in the control unit, say. There may be advantages to having the bulk memory explicitly managed by the software. The bulk memory could be managed either synchronously (where the CPU waits idly upon a page fault) or asynchronously (where the CPU begins processing another task at a page fault). Many questions remain and we refer the reader to [8] for a list of them.

3. On the Organization of Memories

In the last section we described the performance of classical centralized hierarchies for large processors. In this section we discuss the implications of our observations for future memory organizations. We briefly indicate, at a very general level, some new problems which will arise and potential areas for future research.

We have seen that:

(1) logical function — ie. intelligence — is becoming cheap,
(2) the memory hierarchy will comprise an increasingly high proportion of total system cost, and
(3) memory management, when performed by an expensive general-purpose CPU, is becoming increasingly time consuming.

A first implication of these observations is that the memory organization is becoming very important in system design: the memory, rather than the CPU, is the resource whose use will have to be optimized. To replace the loss in CPU performance implied by (3), multiprocessor systems are appearing. Here, two or more CPUs share at least part of the same memory hierarchy.

A second implication is the fact that the CPU could be assisted in some of its management chores by specialized processors. Being specialized, these should be cheap and cost-effective. This trend is already being felt in large computers. For example, I/O processors now provide, asynchronously, much of the function that used to be provided by the CPU. However, they are still very closely monitored by the CPU and could become even more independent.

Other functions such as memory control, data staging, resource scheduling, network communication and accounting could also be offloaded to specialized processors.

What we are observing, then, is the developing need to distribute processing in the memory hierarchy. Numerous authors have observed this — see for example [2]. (The need for distribution can be motivated by phenomena other than performance: for reliability, and growth potential there should be parallelism and modularity within the memory organization.)

Of course, many difficulties are associated with the distribution of intelligence. There are two fundamental design problems. The first is one of how to decentralize. This involves deciding which functions of the CPU are suitable candidates for offloading and what sort of processors should execute them. The second, possibly harder, problem is one of communication and control. The various components must receive their orders from one another and must know something of what the others are doing in order to make informed decisions. So, with which processors, and by what process should they communicate? It must be ensured that the system as a whole, while cooperating, is suitably controlled. Thus, sophisticated storage controllers will have to manage communication links. In this interconnecting system, what should the switches and distributed message systems look like? The difficulties with communication overheads and with sharing that appear in current MP designs only hint at the problems.

Computer designers have little experience in the subject of decentralization and control. The questions of how to decentralize and of how and what to communicate are relevant also to organizations that exist in a variety of other disciplines, and we might hope to draw upon the experiences of others. Towards this end, let us briefly apply a formal approach to the understanding of organizations which has been taken by Simon [7].

Simon defines an "organization" as an assemblage of interacting components that deals in an organized way with its environment. A particularly interesting class of organizations is that of "hierarchies" — these are comprised of interrelated subsystems, each of which is in turn hierarchic in nature. Thus, in an hierarchy there are successive sets of subsystems, with a relationship of subordination and possibly varying "degrees of interaction".

Two views of the memory organization will clarify these definitions. Consider the "memory hierarchy" discussed in Section 2. A conceptual simplicity resulted when we considered the following successive sets of subsystems:

{CPU}, {Cache, CPU}, {Main memory, Cache, CPU},
{Disk, Main memory, Cache, CPU}

In what follows we shall, however, be more interested in a second view of the hierarchy, one which emphasizes aspects of control. Present machines have the following structure:

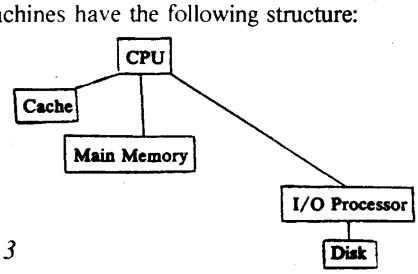


FIGURE 3

Each of the CPU, cache, main memory, channel and disk is itself a subsystem. The diagram indicates that the relationship between the CPU and the cache is more intense than that between the CPU and Main Memory or between the CPU and the I/O processor.

Given these views of the organization, Simon [7] makes some interesting qualitative statements about hierarchic organizations. Based upon empirical and theoretical grounds he has convincingly argued the following.

General organizations evolve most rapidly from simpler systems if there are "stable intermediate forms". The resulting organization will then be hierarchic. Thus we can expect that future decentralized organizations of memory will still be hierarchic. As they evolve over time, intermediate forms (as in the diagram above) will be stable — ie. will function satisfactorily in their own right.

One can distinguish between the internal dynamics of the subsystems and the external dynamics among subsystems. In hierarchies, the internal dynamics are of high frequency, whereas the external dynamics are of low frequency. For instance, the internal dynamics of the CPU and the I/O processor are of different orders of magnitude from the dynamics of their interactions.

Hierarchies, then, are decomposable — the interaction (or communication) between subsystems is weak. It can be derived that the short-run behaviour of each of the components of these systems is approximately independent of the short-run behaviour of other subsystems. In the long run, the behaviour of any one component depends only in an aggregate way on the behaviour of other components. This decomposability of hierarchies greatly simplifies the task of understanding and analysing their behaviour.

In order to achieve high dynamics, the subsystems must specialize in specific functions. The communication (or interaction) between subsystems will be low if the functions they execute are repetitive and predictable. Communications should be standardized; in computers, buffers play an important role in permitting these interactions. Because there are limits to the simultaneous interaction of large numbers of subsystems, there is a limit to the number of subsystems into which a particular component can be partitioned. Contention for resources (such as memory modules and data paths) will restrict the communication between components. So, once again, we have that problems of communication and interconnection will be of major importance. These problems of organizational management, rather than those of optimization of pure horsepower, are notoriously hard to formulate. The brief comments made in this section indicate some desirable properties of an organizational structure and a few of the phenomena that affect it.

4. Conclusion

We have discussed some aspects of memory hierarchies for large machines and have argued the likely directions that future developments will take. Using very simple techniques we have analysed the performance of the levels of the hierarchy and how this performance is affected by various parameters and design decisions. We have seen that the memory is becoming an increasingly expensive resource and that a larger effort will be required to manage and support it. New understanding of the organization of memory, and of topics such as its decentralization and control, will have to be developed.

Of course, there remain many topics that we have barely, if at all, discussed. For example, important issues are the sizes of the memory levels, methods for managing them, and questions on addressability, reliability and volatility. In an attempt to make some general comments we have avoided a host of important details.

Acknowledgements

Very helpful discussions were held with I. Wladawsky-Berger, B.T. Bennett, M.L. Blount, L.W. Hoevel, R.P. Fletcher and many others.

References

1. BLOCH, E. and GALAGE, J.D. (1977). Component Progress: Its effect on High Speed Computer Architecture and Machine Organization, in *High Speed Computer and Algorithm Organization,* Ed. by Kuck, J.D., Lawrie, D.H., and Samuels, A.H., Academic Press, New York.
2. GONZALEZ, M.J. (1978). Future Directions in Computer Architecture, *Computer,* 11, 54-62.
3. KLEINROCK, L. (1975). *Queuing systems, Vol. 1,* Wiley, New York.
4. MATICK, R. (1978). *Computer Storage Systems and Technology,* Wiley, New York.
5. REISER, M. (1976). Interactive Modelling of Computer Systems, *IBM Systems Journal,* 15, 309-327.
6. SCIENTIFIC AMERICAN (Sept. 1977). *Microelectronics.*
7. SIMON, H.A. (1962). The Architecture of Complexity, *Proc. Amer. Philos. Soc.,* 106, 476-482.
8. SMITH, A.J. (1978). Directions for Memory Hierarchies and their Components: Research and Development, *IEEE Comp. Soc.* Int. Comput. Software and Appl. Conference, 704-709.
9. WILHELM, N.C. (1978). A General Model for the Performance of Disk Systems, *J. ACM,* 24, 14-31.

A High-level Programming Language for Interactive Lisp-like Languages

Stef W. Postma

Department of Computer Science, Rand Afrikaans University, Johannesburg, South Africa

Abstract

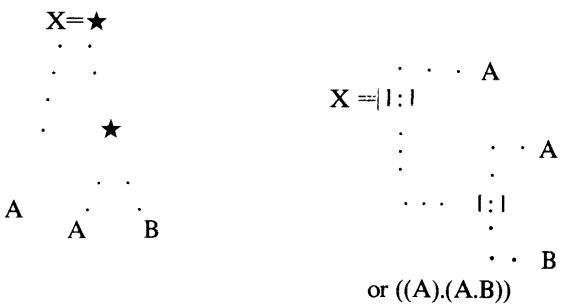
The language LISP 1.5 is neither easy to write nor to read. In the McCarthy et.al. definitive manual on LISP 1.5 a higher level language, MLISP, is used. From the early 1960's various people have proposed higher level languages for LISP. Two types of languages may be distinguished: those based on a LISP programming style, but more readable, and those based on an Algol style. The language PLISP is a language in which it is attempted to reconcile the two approaches to high level languages for LISP. The notation is based on the Algol-style languages but is brief and adapted for terminal use. PLISP allows full access to all the LISP features in a consistent notation. In addition there are features that will appeal to the mathematically mature programmer: full infix notation for not only the assignment, arithmetic, and logical operators, but also for lists, sets, and chained relations. PLISP also has a consistent notation for the first order predicate logic and a variety of ways in which functions may be defined. Top-down program development is facilitated by the use of an assertional language FEA which can be translated to PLISP for test-data generation. Furthermore a definite notation for function stubs is made available — the stubs yield values although the functions called have not been defined at that stage.

1. Introduction: LISP

"... LISP is a formalism for describing algorithms, for writing programs, and for proving properties of algorithms."

John Allen, Anatomy of LISP [1]

The language LISP-acronym for LISt Processor — was defined in the early sixties. Many extensions and modifications to LISP have been proposed over the years, and there is currently no standard for LISP. The language is however, of such beautiful simplicity that a user can find his way quickly in most of the LISP dialects. The power and elegance of LISP makes it an ideal tool for investigating algorithms on a variety of data structures. The basic LISP data type is the leaf-named binary tree — but general graphs may be represented as well. An example of a leaf-named binary tree, X, is:



A LISP function is written in the form:
(function argument . . . . . argument). The basic LISP functions are:

CAR — mnemonically: Climb-Across-Right
eg. (CAR (CAR X)) is A.

CDR — mnemonically: Climb-Down-Right
eg. (CDR(CDR X)) is B.

CONS — inserts its arguments into a bi-pointer (i.e. bip) element. A bip is the computer equivalent of a node in a binary tree.

QUOTE — used to refer to an item instead of the value of the item. E.g. the value of X is obtained by evaluating:
(CONS (CONS (QUOTE A)())(CONS (QUOTE A) (QUOTE B)))

The basic predicate functions are:

ATOM — to test for elementary items,
e.g. (ATOM (CAR X)) is false
(ATOM (CAR (CAR X))) is true.
If Y is not an ATOM then Y equals (CONS (CAR Y) (CDR Y))

EQ — tests its two arguments for identity.

NULL — tests the NIL (or f) value.
e.g. (NULL (CDR (CAR X))) is true.

COND — The LISP equivalent of the IF statement is actually closer to a CASE construct.

The LISP language not only illustrates most of the issues of computer science, but also is extensively used in Artificial Intelligence and Symbol Manipulation systems. The work and research of Artificial Intelligence is usually either implemented in LISP, or in a LISP-based system, where the specialized A.I. language is translated to LISP and the programs executed by the LISP interpreter.

2. The MLISP Languages

The original definition of the language LISP 1.5 is contained in the LISP 1.5 Programmer's Manual by McCarthy et al [4]. There are however no standards for LISP and the implementations of the language contain either modifications or extensions and therefore define dialects of LISP. LISP is a language that is popular and important in artificial intelligence and in symbol manipulation in spite of the communication difficulties caused by the existence of LISP dialects.

The work reported here was done when the author was registered for a Ph.D. at the University of the Witwatersrand.

A more serious problem is caused by the LISP syntax. A LISP program is written in the form of a list so that programs and data have the same syntax: LISP is a high-level language in which programs can be modified or even computed. The syntax of LISP is such that a LISP program is rather unreadable to a human. Consider the definition of the factorial function:

(DEFN (QUOTE FACTORIAL) (QUOTE (LAMBDA (N)
(COND ((EQUAL N 0)1)
(T(TIMES (FACTORIAL (DIFFERENCE N 1)))))))

The discerning reader may have noticed that the parentheses do not balance! A parenthesis balancing technique that has been implemented is the use of square brackets [4], a right square bracket closes all left parentheses up to and including a left square bracket.

The requirements that must be met to improve the readability of LISP are:

1. Parentheses must be kept to a minimum.
2. Clear distinctions between functions and arguments.
3. A natural way of grouping and indenting statements
4. Clear indication of control functions and blocks

The need for a LISP-based language for human communication was realized from the beginning. A language called MLISP — for meta-LISP- is defined in the McCarthy manual. The rules of MLISP are few but the language never caught on. The MLISP form is easier to read than the original symbolic expression, but it hardly meets the three requirements for readability. Higher level languages for LISP followed one of two lines of development: On the one hand more verbose languages in the style of Algol have been proposed: Words like FOR, DO, WHILE, IF, THEN, ELSE are used for control structures. An early effort is the ALISP of Henneman [2]. A currently available language in Algol style is CLISP of Teitelman. The CLISP language is available in the Interlisp system [11],and is not only a high-level language for LISP but it also incorporates testing and pattern-matching features. The second line of development for high-level LISP languages follows on the MLISP initiative. The control structures are indicated symbolically rather than verbally, but a fair amount of verbiage may still occur. Two examples of these developments will suffice, one by McCarthy [3], in Algol style, the other form a recently published book by Allen [1], in MLISP style.

The function considered is the definition of the EQUAL function for LISP. The EQUAL function yields the value t (i.e. TRUE) if its two arguments have the same structure, and are identical at the lowest (atomic) level.

1. McCarthy: Using **at** for ATOM, **a** for CAR, **d** for CDR

```
equal [x,y] ← if at x
                then if at y then x eq y
                     else F
                else if at y then F
                else if equal [a x, a y]
                     then equal [dx, dy]
                else F.
```

2. Allen:

```
equal [x;y] < = [atom [x] →
                   [atom [y] → eq [x;y]; t → f];
                 atom [y] → f;
                 equal [car [x]; car [y]]
                   → equal [cdr [x]; cdr [y]];
                 t → f]
```

The readability of the above two should be compared to the PLISP example below:

3. PLISP: Using α for ATOM, == for EQ, Γ for CAR, $\lrcorner$ for CDR

```
EQUAL (X;Y) ← [?]
              ? α(X) → [?]
                       ? α(Y) → X == Y;
                       ? ⊤ → ⊥;
                       ?]
              ? α(Y) → ⊥;
              ? EQUAL ( Γ X; Γ Y) → EQUAL
                                    (⌟ X; ⌟ Y);
              ? ⊤ → ⊥;
              ?]
```

3. A review of PLISP

The language called PLISP (acronym for Publication/Programming LISP) was developed to facilitate communication in LISP. The improved communication is not only between man- and machine (programming) but also human communication of LISP programs (publication.) The symbols used in PLISP are available on any computer terminal that has APL facilities. The language PLISP has otherwise no connection with APL. The restriction of the symbols to the APL character set was suggested by noting the availability of the symbols to the user.

The PLISP notation is as concise as possible, but a degree of redundancy has been built into the language. This is in accordance with Wirth's dictum [13] that languages may —

'Through their very conciseness and lack of redundancy elude our limited intellectual grasp'.

The level of redundancy was found by experimenting with programs written in the various proto-PLISP languages that were designed. The test for acceptance was simple: at the stage where sufficient redundancy was added to make the 'pattern' — the 'layout' — of a program obvious to the user further additions were stopped. The use of indentation and comments form part of the modern techniques of programming, and are provided in PLISP. Moreover, facilities for topdown refinement and program development are provided by provision for program assertions and function stubs that can return values although the functions have not been defined.

LISP is used in two areas: artificial intelligence and symbol manipulation, and the PLISP that is described in this paper has been devised for the symbol manipulating LISP languages. Extensions for the LISP dialects, e.g. QA4 [9] that have been developed for artificial intelligence are being investigated.

The more noticeable features of PLISP are

1. A natural and readable notation for programming. The notation includes the commonly used infix operators, and clearly indicated control-block structures.
2. The control-block structures that consist of more than one statement has each statement in the block clearly marked. These markers are used as prompts by the computer on a terminal system.
3. The functional notation used in programming has been extended to allow for the notations that mathematicians use. Functions may be prefixed or postfixed to their arguments; functional composition is allowed, and a function may be the value of a calculation. The definition of finite functions are also allowed in an almost mathematical notation.
4. Expressions in predicate logic are allowed and are translatable to programs.
5. Data-constants of various types: sets, trees etc.

The relation of PLISP to LISP is quite simple. The semantics of PLISP is given in terms of LISP by means of a simple Syntax Directed Translation Schema [7]. All the LISP functions are

therefore available to the PLISP programmer. In particular no general input and output functions are defined in PLISP since this will vary with the target LISP dialect that may be used.
Planned extensions for PLISP include:

1. An incremental interactive compiler to Lyspe — a LISP dialect with proper I/O and string manipulation features.
2. Extensions to allow for:
   2.1 Features for artificial intelligence;
   2.2 Database facilities;
   2.3 Syntax directed input
3. Program verification and testing based on the assertional language FEA [6].

All the features of PLISP cannot be discussed in this paper but enough constructs are described to give the reader a working knowledge of PLISP. An example of a PLISP program is given in appendix A, and the reader may find it amusing to read through the example before he reads the description of the language — this will enable him to judge the extent to which PLISP succeeds in being not only a programming language but also a language for human communication.
Some terminology has to be defined. The functions that are of type EXPR, or SUBR, i.e. those that receive their arguments in evaluated form are called the EA functions (Evaluated Argument). The functions that are invoked, and then may, or may not, evaluate their arguments are called NEA (Non-Evaluated Argument)-functions.

4. Control Blocks

A control block is required whenever a number of statements are to be considered as a unit. To facilitate interactive programming, and for the sake of consistency, the control blocks of PLISP are all written with the same basic syntax: control block header in which variables may be named for use in the block, the body consisting of one or more statements each preceded by the control block prompt (or marker), and the control block exit. A prompt must be followed by a blank, a colon or a right square bracket.
Nested blocks may have additional identification on their prompts but this is not discussed in this paper.
A PLISP session at the terminal is initiated by the PROGRAM-header: [□ cset-variables] followed by any number of statements which are evaluated immediately after input. After evaluation the computer prompts the next statement by typing □. The end of a session is signalled by the program exit symbol: □]. The cset-variables are only used for exclusively global variables: these variables may not be used as local variables. Values are assigned to them by means of the CSET function. The variables are initialized to the value ⊥ (NIL) at block-entry time. Most programmers will not make use of the CSET variables, but they are required for LISP systems that make use of the APVAL-value of a variable.

```
E.g.                                  Typical translation:
[□A ; B]                              (PROGRAM (A;B)
□: CSET ('A;5);                       (CSET (QUOTE A) 5)
□: CSET ('B;A∘ ∘A);                   (CSET(QUOTE B)(CONS A A))
□: □B;                                (PUTDATA B)
□]                                    )
```

The other control structures are:

1. The CONDitional corresponding to the case statement.
2. The PROG corresponding to BEGIN . . . . . END.
3. The REPEAT corresponding to various kinds of loops.

Local variables may be named in each of the control structures. Such variables are relatively global to any contained blocks, unless they have been named again. The local variables are always initialized: if at block entry time there is a relatively global variable with the same name then the local variable is initialized to the global value, otherwise the local variable is initialized to ⊥(NIL). The values of the local variables are lost at exit from the block.
The conditional, with its prompts, is of the form:

```
[? local variables]
? p1 → e1;
:
:
? pn → en;
?]

E.g. [? A;B]
     ? 'A: = ⌊ B → ⌈ B;
     ? ⌊ A → ⌈ ⌊ B;
     ? ⊤ → B;
     ?]
```

which translates to:

```
(COND (A;B) ((SET (QUOTE A) (CDR B))
  (CAR B)) ((CDR A)(CADR B)) (T B ))
```

or, if the target LISP does not allow local variables in a COND, the translation is:

```
(PROG (A;B) (COND ((SET (QUOTE A) . . .
```

If none of the pi have a non-nil value then the conditional is, typical LISP-language like, undefined.
The PROG, with its prompts, is written:

```
[ω local variables]
ω statement;
:
:
ω statement;
ω]
```

The statements of a PROG are evaluated from the first statement to the last statement (i.e. from alpha to omega). The value of the last statement is the value of the PROG — hence the mnemonic symbol ω (omega) used as the PROG symbol. Labels and a GO-function are not defined in PLISP. If however they are really required, the LISP coding may be used in the PLISP program to achieve the desired effect. LISP coding may be entered at any point by enclosing that coding in the special quotingbracket: slash (/).
The repeat (loop) construct must have an exitcase as one of its statements. The exitcase determines the value of the repeat, and replaces the COND with a RETURN of LISP. Furthermore, the exitcase acts like the COND in a PROG of LISP: If no condition of the exitcase is satisfied then the evaluation continues with the statement following the exitcase.
A repeat block is written:

```
[ρ local variables]
ρ statement - 1;
:
:
ρ statement -n;
ρ [ε local variables]
   ε p1 → e1;
    :
    :
   ε pj → ej;
   ε]
ρ statement -a;
:
:
ρ statement -z;
ρ]
```

Either or both of the sequences of statements 1-n or a-z may be empty.

5. Infix Operators and Data-Types

It is customary to assign a precedence among infix operators, for example a + b × c is usually interpreted as a + (b×c). An operator furthermore is usually interpreted as being left associative, i.e. a+b+c is evaluated as (a+b)+c. In PLISP the operators are classified as:

LA — left associative eg. A+B+C is (A+B)+C
RA — right associative e.g. A:=B:=C is A:=(B:=C)
MLA — mutually LA e.g. A-B+C-D is ((A-B)+C)-D
MRA — mutually RA e.g. $A_{oo}B_oC_{oo}D$ is $A_{oo}(B_o(C_{oo}D))$
LAC — LA-chain e.g. A =B = C is (A= B) ∧ (B = C)

The operators and typical data constants are given in the table below. The entries have low precedence at the top to high precedence at the bottom.

Infix Operators

Operator	Type	Lisp-name	Data constant e.g.
←		DEFUN	
:=	RA	SET	
<=	RA	SETSTRING	··String const··
oo	RA	CONS	
o	RA	CONC, APPEND	/S-expression/
ρ	RA	PAIR	
~	RA	NOT	
∧, ∨	MLA	AND, OR	⊥ is 'false'
⊃	LA	IMPLIES	T is 'true'
⊃⊂	LAC	EQUIV	
<,>,=, ≠, ε, ⊂, = =,	MLAC	EQ	
/ /,		SUBSTRINGP	
⊂⊃		SUBSETP	
∩,∪	MLA	INTERSECTION, UNION	
¯	LA	SETCOMPL, relative	
★	LA	CARTESIAN	{X predicate}
2★	RA	POWERSET	{X ε S predicate}
+,−	MLA		integer
× ÷	MLA		fixedpoint
↑			floating point
	LA	CATENATE	
↓	LA	SUBSCRIPT	
=>	LA	INDIRECT addressing	
	RA	CDRASSOC	≤edge-named tree≥

6. Prefix, Infix, and Suffix Functions

Three types of function names may be used in PLISP. Firstly any LISP function may be used, secondly a number of standard LISP functions are represented by symbols, and finally the user can define his own functions (and operators). The defined functions may have any valid name, or the name may be formed from a prescribed set of symbols. The predefined function symbols are all used as prefix functions. If they take a single argument then that argument need not be enclosed in parentheses if the argument is an identifier. The predefined function symbols are:

E.g.	translates to:
'A	(QUOTE A)
⌈ B	(CAR B)
⌊ C	(CDR C)
□ E	(PUTDATA E)
αF	(ATOM F)
⊥ G	(NULL G)
εαH	(EVALA H)
ρA(I;J)	(RPLACA I J)
ρ D(K;L)	(RPLACD K L)

The function defining functions are:
ΔE (fname genlambda) for EA functions
ΔN (fname genlambda) for NEA functions
ΔT (fname genlambda) for temporary EA functions

Function calls may be specified in various ways. The first type of call — symbolic function followed by identifier-argument — is given above. The most common type of function call is of course: function-name (arguments) ⌊ which is also given above. A number of other ways of invoking a function are also allowed, they are:

1. Post-fixed function calls
2. Compound function calls
3. Calculated function calls
4. Tabular function definition
5. Generalized lambda functions

In a post-fixed function call the arguments precede the function. In this case the arguments are enclosed in a special pair of brackets, i.e.: [(arguments)] function-name.

A compound function call is the PLISP equivalent of the mathematical composition of functions. The form F(B(H . . . (M(args). . .) may be written as: [o F ; G; H ; . . . ; M o] (args)

A function is represented by a lambda-expression in the LISP languages. A function name is used by the EVAL-routine to establish the particular lambda-expression that defines the function. In LISP therefore a function — be it name or lambda-expression — may be calculated. Since a calculation may be any form a special notation is needed to distinguish the calculated functions from other forms. A calculated-function call is therefore written in the form:

[ιexpressionι] (args)

A function is always defined, even if only implicitly (in PLISP that is) in terms of a lambda-expression. A lambda expression is specified by: [ν proto-arguments] expression ν]. The proto-arguments are the identifiers that are used as local variables in the expression. The proto-arguments take on the values of the arguments on the systems a-list at function invocation. They are either simple identifiers, or are identifiers prefixed with an iota-symbol — the latter are called inhibited proto-arguments. The inhibited proto-arguments are not stacked but only modified on the systems a-list if an inhibited recursive function call is encountered at time of evaluation. An inhibited function call is specified by prefixing the function name with an iota. This allows recursive functions to be evaluated wholly or partially iteratively.

The tabular function definitions are used to define EA functions according to standard mathematical notation. An example suffices:

ADD5: INTEGER ★ INTEGER → INTEGER:
[★0 1 2 3 4
★0→0 1 2 3 4
★1→1 2 3 4 0
★2→2 3 4 0 1
★3→3 4 0 1 2
★4→4 0 1 2 3 ★]

Expressions in the first-order predicate logic are also interpreted as implicit lambda expressions. The functions corresponding to these expressions take only sets or integers as arguments. The expressions allowed are quantified formulae, and the first quantifier must be a restricted quantifier [8], which specifies the universe of discourse. The other quantifiers may be restricted or not depending on whether they refer to another or the previously specified universe.

The unrestricted quantifiers are:
[∧ X] i.e. for all X
[∨ Y] for some Y
[∨1 Z] for one and only one Z
The restricted quantifiers:
for sets: [∧ X∈S] i.e. for all X of S
[∨ Y∈T] for some Y of T
[∨1 Z∈U] for one and only one Z of U
for integers: [∧ X < N] for all X less than N, X ≥ 0
[∨ Y < M] for some Y less than M, Y ≥ 0
[∨1Z <P] for one and only one Z less than P, Z ≥ 0.
E.g. [∧ X ∈ S] [∧ Y] [∨ W ∈T] P(XYW) ∧]
In the above expression X and Y range over the same universe S and W ranges over the set T. The quantifier expression is translated to a lambda-expression with S and T as proto-arguments. In the body of the lambda expression, the X,Y and W are defined as local variables. During execution of the function, X and Y will range over the set which is the argument corresponding to S, and W over the set which is the argument corresponding to T.

7. Top-Down Development: Assertions and Stubs

There are three requirements for top-down programming: the first is a language facility to enable the user to express overall, or top-level, program constructs. The second is a facility for defining program or function stubs: procedure calls to routines that are yet to be written. The third is a language facility to enable the programmer to make statements or assertions about the program.
Assertions may be used in PLISP programs in addition to comments. The body of the assertions are written in a language called FEA which has been described elsewhere [6]. The FEA-assertions are English-like statements in the predicate logic that can be translated to PLISP, and may be used as input to a program verifier.
An assertion is written in the form:

[α local variables]
α FEA assertion
:
α FEA assertion
α]

Function stubs are calls to functions that have not yet been defined. The called function, until it is defined, usually does only some simple-minded thing like return a constant value or prints out a message e.g. 'FN AX3 CALLED'. The function stubs that are possible in LISP languages can be much more powerful since an identifier can have a value and also be defined as a function. The FNSTUB function then tests its function argument to see whether it has been defined as a function, if so FNSTUB calls that function; if the function of the functionstub has not yet been defined, then it must have a list of values as its value. The FNSTUB returns the CAR of the list as the value of the functionstub call, and sets the list of values to the CDR of the list.
In PLISP a functionstub is written:

[Δ id: id; . . . ; id/id : id; . . . ; id/ . . .]
optional assertion
function (arguments)
Δ]

The - id: id; . . .; id - are declarations and may be used in any way by the programmer. For example, data types may be declared, or variables may be declared as global in the function to be defined.

References

[1] ALLEN, J.R. (1978). *Anatomy of LISP,* McGraw-Hill, New York.
[2] HENNEMAN, W. (1964). An Auxiliary Language for more Natural Expression — The A-Language, in Berkeley, E.C. and Bobrow, D.G. (Eds). *The Programming Language LISP : Its Operation and Applications,* M.I.T. Press, Cambridge, Mass.
[3] McCARTHY, J. (1977). Another SAMEFRINGE, *SIGART Newsletter,* No. 61, Feb. 1977, p.4.
[4] McCARTHY, J., et. al. (1965). *LISP 1.5 Programmer's Manual,* 2nd ed., M.I.T. Press, Cambridge, Mass.
[5] MANNA, Z. and WALDINGER, R. (1977). *Studies in Automatic Programming Logic,* North-Holland, New York.
[6] POSTMA, S.W. (1978). FEA — A Formal English Subset for Algebra/Assertions, *SIGPLAN Notices,* **13,** 7, 43 - 59.
[7] POSTMA, S.W. (1977). Some Useful Techniques for Defining Formal Languages — With examples from FEA and PLISP, paper read at 1977 Symposium of the Computer Society of South Africa
[8] ROSSER, J.B. (1953). *Logic for Mathematicians,* McGraw-Hill, New York.
[9] SAMMET, J.E. (1969). *Programming Languages: History and Fundamentals,* Prentice-Hall, Englewood Cliffs.
[10] SANDEWALL, E. (1978). Programming in an Interactive Environment: The 'LISP' Experience, *ACM Computing Surveys,* **10,** 1, 35 - 71.
[11] TEITELMAN, W. (1975). *Interlisp Reference Manual,* XEROX, Palo Alto.
[12] UNIVERSITY OF WISCONSIN COMPUTING CENTER, *LISP Reference Manual for the UNIVAC 1108,* UPLI Reference No. 800022.
[13] WIRTH, N. (1974). On the Design of Programming Languages, in *Information Processing 74,* North-Holland, Amsterdam, pp. 386-393.

Appendix A

```
*** Definition of the AND function. Symbols used: ΔN -
  DEFNEA;
* ΔE — DEFEXPR; ∇ — LAMBDA; ⊥ — NULL, NIL; ⊤ —
  TRUE; εα — EVALA;
* ι — Inhibitor, ⌜ — CAR; ⌞ — CDR; ' — QUOTE.
* AND is a NEA, LA function, and it is the PLISP equivalent of
* the logical conjunctive. If it has no arguments then its value is ⊤.
*    If it has one argument then the value of that argument is
* a list of items to be ANDed. Else its arguments are evaluated
* from the left. ***
ΔN('AND '[∇PA]
[?] ? ⊥PA →⊤;
   ? ⊥(⌞PA)→ ANDEA(εαPA:⊤);
   ?]Δ])
   ΔE ('ANDEA '[∇ ιPB; ιPC] *** Auxiliary Function ***
   [?] ? ⊥PB→⊤;
   ? PC →[?] ? [PB →ANDEA(⌞PB;PC);
            ?⊤→⊥;
            ?]
   ? ⊤→[?] ? εα(⌜PB)→ιANDEA(⌞PB;PC);
            ? ⊤→⊥;
            ?]
   ?]∇]
   *** alternative definition using iteration ***
   ΔN ('AND '[∇PA]
        [?] ? ⊥PA→T;
           ? ⊥(⌞PA)→[ω]ω'PA:=εαPA;
                      ω[ρ] ρ[ε] ε ⊥PA→⊤;
                              ε~(⌞PA)→⊥;
                              ε]
                           ρ'PA:=⌞PA;
                           ρ]
                      ω];
            ? ⊤→[ρ]ρ[ε] ε ⊥PA→⊤;
                      ε~(εα(⌜PA))→⊥;
                      ε]
                 ρ 'PA:=⌞PA:
                 ρ]
            ?]∇])
```

Thirty Years of Information Engines

G. G. Scarrott

ICL Research and Advanced Development Centre,
Stevenage SG1 2DX, England

Abstract

The era of information engineering was initiated some thirty years' ago by the demonstration of the first stored program electronic computer. At that time there was intense innovative excitement among the pioneers which, however, became somewhat jaded in the subsequent decades as computers became a business and the business acquired its doctrinaire echelons of orthodox systems experts.
More recently, the excitement has begun to return as the triumphant progress of LSI technology has reopened the frontiers of systems engineering so that it is now appropriate to review the evolution of computers with the object of gaining an appreciation of where we are now, by what route we arrived, and what can be expected to happen next. The paper includes brief reference to research into new systems concepts, relevant to such a forecast that are now moving into the market place.

1. Introduction

It is generally understood that the era of information engines started about thirty years' ago with the demonstration of the first stored program electronic computer. At that time, there was intense innovative excitement among the pioneers which, however, became somewhat jaded in the subsequent decades as computers became a business and the business acquired its doctrinaire echelons of orthodox high priests.
More recently, the excitement has begun to return as the triumphant progress of large scale integrated device technology has re-opened the frontiers of system engineering. I therefore propose to outline the evolution of computers with the object of gaining an appreciation of where we are now, how we arrived, what can be expected to happen next and some new system concepts relevant to such a forecast that are now moving into the market place.
A clearly discernable feature of the evolutionary process has been a somewhat fitful trend towards regarding computer system design as an engineering discipline in the conventional sense, characterised by an understanding of design objectives, concern with cost effectiveness, and control of implementation techniques by a subtle combination of theory and practice. Thus, the scope of this paper is probably best described as "the evolution of information engines" — perhaps an unfamiliar phrase but it has the advantage that it not only covers "computers" as we have come to understand the term but also includes foreseeable developments which have not yet occurred.

2. Technological evolution

Let us consider the technological evolution of a typical product in the four phases: birth, adolescence, maturity and senility. When the new product first appears the technology for making it is primitive but, if it serves a useful purpose at all, pioneering users exist whose needs for the new product are so pressing that they are willing to adapt their practices to take advantage of it.
Thus, in the beginning most of the discussion is concerned with "how" to make and use the product and there is very little discussion of "what" purpose the product should serve or "what" should be its technical specification to serve such a purpose. In the adolescent stage, the main population of users begins to appear. Many of these do not have such a clearly recognised need and, indeed, some of them may be only following fashion. As a result of this, discussions regarding the new product begin to tackle the more fundamental issues of "what".
Nevertheless, the technology is still immature so that the adolescent stage can be roughly characterised by the fact that "how" and "what" debates occur about equally, often with regrettably little cross-fertilisation between them. When the product is mature, the technology is fully developed and the market is saturated. All concerned know perfectly well that any relevant product specification can be made so that the crucial questions are entirely concerned with framing a specification which will attract enough users to justify the business. Thus, at this stage, "what" is dominant. Finally, when the product becomes senile it goes out of use and its social purpose is met by other products.
Diagram 1 summarises this maturing process as applied to steam engines, aircraft engineering and information engines. It suggests that information engines are still in the adolescent stage.

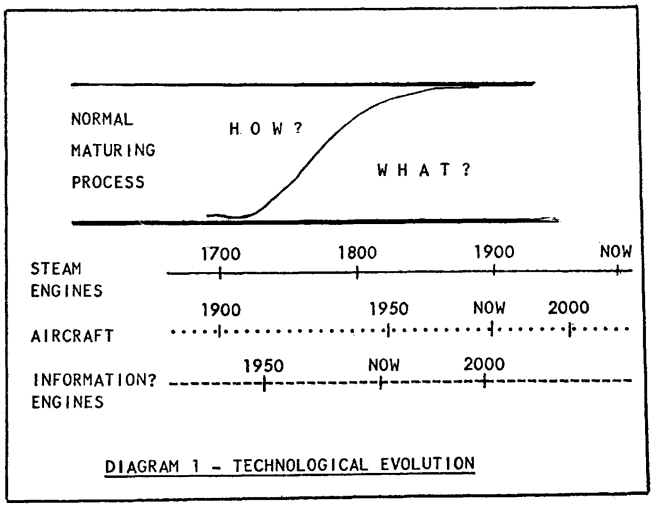


DIAGRAM 1 - TECHNOLOGICAL EVOLUTION

DIAGRAM 1

The first useful electronic information engine was developed during the second World War. Since that time, the physicists and electronic engineers have done a splendid job introducing solid state devices and large scale integrated fabrication techniques which have removed out of sight many of the technological constraints which shaped the early information engines. However, the refinement of technology has not yet been complemented by an understanding of the natural properties of information adequate to guide the deployment of our new found technological mastery so that a first approximation to an understanding of the present situation in information engineering would be to liken it to the situation in the evolution of steam engines in the early nineteenth Century after techniques for casting, forging and machining had provided the "means" but before theoreticians such as Carnot and Rankine had illuminated the "ends" for steam engine design.

3. The present situation in information engineering

Against the foregoing background of general technological evolution, it is useful to summarise the present situation:-

(i) Centralised Data Processing is common.
(ii) Small decentralised data processing systems are proliferating. These contrasting styles of operation are already being widely discussed.
(iii) Interactive use is costly and fragile.
(iv) Complexity is difficult to control. The complexity of an assembly of hardware or software is essentially measured by the quantity of information required to describe it. Complexity is not the same as multiplicity. Thus, for example, if we look at a semiconductor storage chip through a microscope we will see many thousands of components arranged in a highly systematic pattern. Such a storage chip represents high multiplicity but low complexity. On the other hand software comprises almost pure information so that it is characterised by pure complexity. Indeed, human limitations in the handling of complexity now control the range of purposes for which information engines can be effectively used.
(v) Processors originally designed for arithmetic are mainly used for other purposes. We have known this for a long time but done very little about it.
(vi) LSI is used to reduce cost and improve reliability. These are proper objectives but LSI has not yet been used to improve basic system designs which have changed very little over 30 years.
(vii) Device companies are beginning to enter the systems business by offering naked hardware without much software support.

4. Constructional technology

So much for the present situation, what is going to shape the future? It is easy to summarise the technological situation. The essence of the matter is that planar micro-fabrication techniques have rendered the constructional units for processor, fast store, and first level backing store so similar that all three elements can now be assembled in any mixture that we need. There is no longer any over-riding constructional reason for continuing the traditional practice pioneered by Von Neumann of concentrating each type of element, main store, backing store and processor in a separate box and interconnecting the boxes via bottlenecks. Thus, technological advance permits us to design our systems to meet our up to date understanding of the users' requirements but it does not tell us how to do this. Only a deep understanding of the interface between the system and its human users can do this.

5. Essential requirements

It is not so easy to summarise the requirement situation. It is no use trying to analyse the tangled web of computer applications — that way leads only to confusion. Neither is it useful to rake over current systems implementation practice since that has been done many times already. We must start at the beginning. To achieve an understanding of the nature of information and its role in human affairs, adequate to guide the design of an information system, the study should be regarded as a branch of biology rather than mathematics or electronics.

We must recognise that the human race is a species whose original survival and present dominance is based on the chance discovery by our remote ancestors of a new field for biological competition — the creation and operation of social groups dynamically adaptable to the environment by large scale interchange of information between individuals and groups.

We still do this on a grand scale. The hunting and agricultural teams have now become "companies" and the day to day operation of the social co-operation mechanism is called "business". Nevertheless, in our business operations we compulsively adopt organization techniques and associated information handling techniques which served our ancestors for a million years, so that when we introduce a computer system we ignore such inflexible human habits at our peril. However, when computers were first introduced into business the constraints imposed by primitive electronic technology necessitated that to a great extent the user adapt his practices to the computer rather than vice versa. For example, the centralisation of information processing and its collection into artificial batches represented such a forced adaptation. These practices have continued to the present day and have led to many frustrations which have contributed to the somewhat ambivalent image of the computer and its professional attendants as essential but awkward.

We can now recognise that the commercial centre of gravity of information engineering is concerned with "Data Base Management" where the meaning of the phrase should properly be derived from the human realities. The purpose of a "data base" is to assist communication between the members of a co-operating group of people by maintaining a continuously up-to-date information image of the current state of the group and its relevant environment. This purpose can be served only if all the individuals associated with the data base, those who put information in as well as those who access it, feel that the data base is of sufficient value to the group and to each individual to justify its cost. In present practice, a typical data base is not always up to date, it permits its users to ask only stereotyped questions and even to achieve this necessitates a mountain of software which imposes an unacceptable parasitic load on the available computer power and poses a formidable software maintenance problem.

6. Natural properties of human information

It is now possible to see clearly that an effective solution to such problems can be devised only by cultivating an attitude of humility in the design of our information systems. We must first recognise that there is an underlying unity in the tangle of computer applications and that it is derived from their common factor — people. Information structures appear at first sight to be arbitrary and ad hoc — invented on the spur of the moment for each specific purpose. However, such structures are heavily influenced by habits which have been evolved over a long period to create and maintain co-operative social groups. One such habit, the use of tree type data structures, obviously derived from the wide use of tree type social organization structures, is well known to the designers of high level

languages but has seldom been reflected in computer design.
A consequential and less obvious habit arises from the fact that for every real situation there are many alternative ways of organizing the associated information in relevant tree structures so that as the situation evolves new tree structures unrelated to those already established tend to be created — grow in importance and then fall out of use. Thus, at any time, many alternative organization structures can be said to have meaning to the people whose co-operation gives rise to the information and it is not practicable to represent all such structures by indexes in their data base.
A third information handling habit arises from our instinctive preference for the interactive mode of information transfer. Although the human species can legitimately be regarded as an information handling specialist and we commonly handle large quantities of information, we can best deal with it in small packets with frequent opportunities for checking and clarification. Accordingly, when we access a data base we reserve the right to alter our question in response to the information obtained. Similarly, we find it impossible to create a large program free of errors, so that every program needs to be debugged before it can be used and, indeed, during use. Curiously, some computer professionals tend to take a puritanical view of programming errors. However, the prevalence of errors is simply the human preference for the interactive mode asserting itself and eventually we can expect that a typical information system will be designed from the beginning to be controlled in the interactive mode.
With this view of the social nature of information, we can deduce some more natural properties:

(i) Most information in the world is not numeric.
(ii) A growing proportion of information processing is not arithmetic.
(iii) Logical operations can be used for all purposes including arithmetic processing.

This has long been known as a mathematical proposition but its practical relevance has only recently been rediscovered and, indeed, we have exploited it in our Distributed Array Processor project. These natural properties of information are summarised in Diagram 2.

Diagram 2

Natural properties of information

1. "Information" bonds society.
2. Information handling habits have been shaped by long use of information to operate social groups, e.g. hunting, agriculture, business.
3. Organization structure of information reflects the structure of the human group which it serves.
4. Hence, Organizational structure of information is normally a blend of order and disorder.
5. People communicate information in small packets with opportunity for checking (interactive mode).
6. Some information processes naturally occur in batches, e.g. payroll.
7. Most information in the world is not numeric.
8. Growing proportion of information processing is not arithmetic.
9. Logical operations can be used for all purposes including arithmetic processing.

7. A technological forecast for information engineering

If we extrapolate from the present situation taking into account the analysis of requirements and available technology which I have outlined, a forecast for the next decade of information engineering is almost obvious. The prime objectives for system design will be data management and keeping complexity under control. The essential technique for ensuring that complexity is manageable is to arrange that the quantity of information handled by the designers or users at one time is within human capability. Thus, the design of both hardware and software must be modular. The modules must not be too large and they must be separated by clean and tidy interfaces such that errors in one module cannot sabotage others. Above all, it will be recognised that universal geniuses do not exist so that the complexity problem cannot be brought under control by seeking more competent people or appointing a new project manager.
A typical system will be designed to be used interactively by people who are primarily interested in carrying on their own business and have no interest in the technicalities of data processing. Batch processing will be confined to operations which naturally occur in batches, e.g. payroll.
At the present time attempts to assess the power of a computer system are somewhat confused by empirical and highly artificial units such as the Post Office Work unit or MIPS. We shall come to recognise that the natural measure of the power of a system is simply the number of people in the organised social group which the system can serve. Systems will be available which will be cost effective serving only a few tens of direct users and they will be purchasable at a cost which will not require user board approval. Such small systems will perform functions which can be clearly understood by the user who will be able to interconnect and redeploy his information engines to meet his constantly evolving needs.
Intrinsically symmetrical communication techniques will be adopted to make this possible. The proliferation of such small systems will take away some of the load of a typical centralised data processing system so that the data processing manager will change his role. Instead of taking direct responsibility for all information processing, he will ensure that the small systems distributed over the users' organisation are compatible with one another and can be used to extract corporate information in addition to serving their primary role as departmental information systems.

8. Some ICL research projects

For several years the work of RADC has been selected to prepare for the situation which I have described. We have put most of our efforts into four projects: Variable Computer Systems (VCS), Content Addressed File Store (CAFS), Distributed Array Processor (DAP), and Interactive Man/Machine Communication by Speech.

8.1 Variable Computer System

The conceptual origin of the Variable Computer System project was the recognition that the natural way that people access information is not by the use of a fixed reference framework as in the von Neumann machine but by the use of a reference map which can be created and continuously maintained up-to-date by the user to suit his purpose. The objectives of the VCS project were two fold:

1. To demonstrate a working system incorporating low level navigation facilities by means of which a user can create and maintain up-to-date a secure reference map showing the organizational structure of his information and its mapping on physical storage.
2. To take advantage of systems incorporating fast microprogram storage by permitting the target machine to be adaptable on

demand to the high level language in which the source program is written.

We now know that the first objective is similar to the capability systems designed at the Universities of Chicago and Cambridge. The provision of a secure map makes impossible many of the programming errors, the unmonitored accumulation of which, accounts for the severe difficulties commonly encountered in the development and maintenance of complex software.

The VCS System has been demonstrable in the Research Department since 1974 and recent work has shown how it could be implemented on standard Company hardware (the MICOS 1 Processor in the 2903).

We have compared the performance of the VCS/MICOS 1 System obeying COBOL programs with the performance of 1900/MICOS 1 (2903) on the same COBOL programs and find that VCS offers:

(i) a reduction in COBOL compiler size in the ratio of 6:1;
(ii) a reduction in COBOL object code size of up to 3:1;
(iii) an increase of COBOL execution speed in the ratio up to 1:1.8;
(iv) a more powerful operating system (in the sense of providing more facilities) which is at the same time more flexible (in the sense of adapting rapidly to different modes of use), with about the same storage requirements;
(v) reduction in main store quotas per job as a result of code sharing and automatic adjustment to working set size.

The complete technique of creating, maintaining, using an information map can be summarised as "access by navigation". This navigation technique is of necessity used in all existing computer systems but in most of them the map is specified completely only in the source version of the program and is irreversibly confused at the time when it is most needed, that is, after compilation. The VCS system preserves the information map explicitly at run time so that accidental deviations from authorised paths on the map can be immediately monitored and controlled. The intrinsic security which these mechanisms permit operate within as well as between programs and serves all the software including the system software, at trivial cost.

Evidently, the use of the navigation technique requires that the work involved in creating and maintaining the map be small compared with the work representing the primary purpose of the information system. This was the case for the tasks for which computers were first used. However, of late it has been increasingly recognised that Data Management can be expected to become the most significant use of information systems. A data base, by its very definition, represents in information terms the activities of people whose interactions cannot be totally predictable. It follows that the maintenance of the information map of a data base involves a very great deal of work to ensure its continued integrity and accuracy as a true representation of the network of agreements, promises and achievements which bind a co-operating group of people. This situation has become widely recognised by those who have been trying to devise standardised navigation techniques for Data Management. Indeed, this recognition no doubt accounts for the controversial nature of standardisation proposals such as CODASYL. This problem is a fundamental consequence of an intrinsically unpredictable component of the natural behaviour of human information so that it is unlikely ever to be overcome within the limitations imposed even by an efficient and secure navigation technique such as used in the VCS system.

It is therefore also necessary to provide a complementary technique to retrieve information by search in those circumstances when the work involved in maintaining the map up-to-date is either excessive or in some cases, intrinsically impossible. There is therefore an unanswerable requirement for a cost effective technique for accessing information by searching for it. The CAFS project to which I have already referred was undertaken to explore ways of providing such a facility and using it in combination with the navigation technique.

8.2 Content Addressable File Store

Over the past few years, we have, in the ICL Research Centre, built such a searching device based upon the use of disk files and we have conducted extensive experiments on methods of using such a facility to meet difficult requirements such as for Telephone Directory Enquiries systems, bibliographic information retrieval, and management information systems. The primitive operations carried out by the searching engine are illustrated in Diagram 3. A selection function is formed from a description of the required data unit issued by a terminal message or by a program. The encoded selector is passed to a backing-store controller equipped with scanning hardware which comprises key-matching channels operating simultaneously on a stream of data, and a special processor which evaluates Boolean or threshold expressions using the comparator outputs as arguments. A wide range of common types of selection function can be represented in a very direct manner in the two-level evaluation hardware.

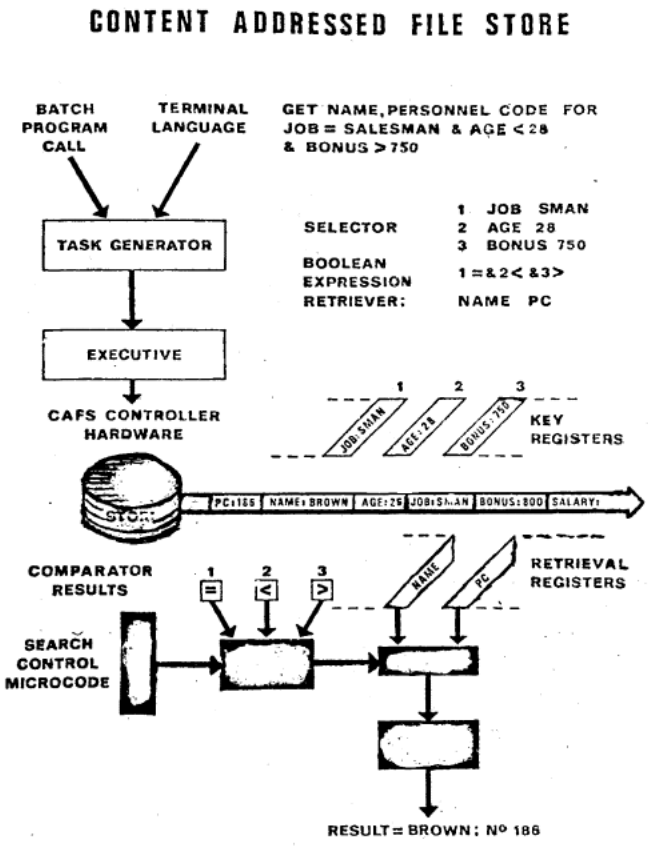


DIAGRAM 3

It is possible for more than one independent search task to be active on the same data stream, the major constraint being the number of key channels available. The latter plug into a standard highway system, and may be thought of as a variable resource analogous to mainframe storage. Each channel is capable of detecting relevant data fields by matching against embedded identifiers, and comparator masking is available to permit part-fields to be isolated. Retrieval of data from "hit" records can be achieved selectively by collecting the contents of designated fields. It is therefore possible to compose virtual "reply records" comprising only that data required by the calling process, arranged in a specified sequence. Alternatively a count of "hits" may be all that is required, in which case, no data as such is recovered. This editing of results is particularly valuable in minimising central resource loading in interactive situations and is generally beneficial in view of the size to which multi-purpose records can grow.

The hardware control facilities are completed by normal physical device controls, including write channel administration. Overall organization is effected by a miniprocessor which is also available to provide a further level of data sieving and composition. The overall balance of search hardware is therefore seen to comprise a filtering mechanism in which the full backing store transfer rate is handled only by very simple, repetitive hardware, with progressively more complex operations being performed on successive abstracts of diminishing volume, culminating in procedures executed in the mainframe. Thus, the sum of the products of data rate and complexity of operation is minimised and, in particular, the mainframe mill and backing store channel load can be reduced by several orders of magnitude compared with conventional serial file processing.

The storage medium can be serial or block-accessed, such as tape or disk. The most generally useful device is the magnetic disk. On a typical high-capacity disk handler many heads are available that require only the addition of some fairly inexpensive electronics to provide a greatly increased read-out rate. Such a high rate would flood all but the most powerful central processor but the progressive abstraction scheme of the special scanning equipment described above renders high speed searching entirely feasible.

Our studies of the use of autonomous file searching devices have shown that it is quite practicable to implement a relational model of a data base. Indeed, we have found it possible to refine the relational model beyond the published work of Codd and his associates.

As I have explained, this project originated in a conscious attempt to identify the facilities required for a Data Base Management system and to provide them by taking advantage of up-to-date technology. We have now reached a stage in which we have studied a variety of specific manifestations of the generalised data base problem and we have not yet discovered any reasons for changing our view of the intrinsic nature of the data base management task.

8.3 Distributed Array Processor

Our third project is the Distributed Array Processor which was initiated about five years ago in direct response to the recognised requirements of weather-forecasting and meteorological research. Since that time, the scope of the work has been broadened to include a very much larger range of problems, e.g. in plasma physics and associative information retrieval.

The great majority of present computer systems can be regarded legitimately as direct descendants of the Von Neumann machine in the fundamental sense that they are characterised by four features:

1. The processor is primarily designed for arithmetic operations with logical operations regarded as a by-product.
2. The store and the processor are separate.
3. The store processor combination can obey only one instruction at a time each requiring not more than two operands.
4. The essential objective in programming such a machine is to represent the overall task by a serial string of such instructions.

These primary features have been somewhat blurred in some powerful machines by tactical measures such as pipe-lined operations on ordered strings of operands but the fundamental principle that strings of individual instructions are obeyed sequentially is still valid. Hence, the connection between store and the processor inevitably imposes a well-defined upper limit to the rate at which the whole assembly can operate. In short, it is a bottleneck.

With the introduction of semiconductor storage and large scale integrated circuits, the original reasons for separating processor from storage are no longer valid. Furthermore, most of the information in the world is not numeric and consequently a growing proportion of computer operations are not arithmetic, so that we should now regard arithmetic processing as a specialised use of more general and fundamental logical operations. Accordingly, to deploy up-to-date technology to meet a broad spectrum of users' requirements, it is now appropriate to revise all four features of established system design practice. In the DAP all these changes have been made.

The conventional semiconductor store is inevitably made in many elements each typically storing a few thousand bits. In the DAP each element has its own very simple processor primarily designed to carry out logical operations on one bit operands. It writes its results into its own storage element and can use as input, information from its own store, its immediate neighbours or elsewhere via row or column highways in a matrix of storage elements. Thus, the DAP offers the following fundamental advantages over current methods.

1. The simple processing elements are easy to design and build, and are flexible in use.
2. The physical distance between each processor can be very short.
3. There can be many such store element/processor element connections operating simultaneously.
4. Real problems are commonly parallel by nature. The DAP provides a parallel processing capability which can match the structure of the solution to that of the problem.
   The DAP-FORTRAN language gives a concise and straightforward way of expressing parallel operations and has already been used to program several applications on the pilot DAP.
5. Since the processing elements are primarily designed to carry out logical operations, a valuable speed up factor compared with present practice is achieved on all operations, data manipulation as well as arithmetic operations. A typical present computer system tends to spend more of its time on data manipulation than arithmetic so that the DAP offers a substantial performance advantage on a wide range of applications.

All the processing elements obey a common program but each element can be instructed or instruct itself to ignore any command in order to provide sufficient flexibility to enable the apparently rigid matrix to be adapted to the parameters of real problems.

The complete assembly can be regarded as a store which has all the properties of a traditional store but with the extra facilities which have been described. It can therefore store its own instructions in the normal way. Moreover, the DAP store can be incorporated as part of the store of an existing host computer of conventional design which is responsible for putting the problem into the DAP part of its own store and getting the answers out. In this way, it is possible to

take advantage of the power of the DAP to tackle difficult processor intensive parts of real problems without requiring complementary development of a new operating system.
The proposal was conceived in 1972 and a pilot 32 x 32 has been working since early 1976. It has now become clear from the use of the pilot that the DAP can be applied to a wide range of information processing tasks and, indeed, that intrinsic serialism in real problems is quite rare. Table 1 shows performance estimates for examples from the indicated application areas. A 64 x 64 DAP has been ordered by the Computer Board for use in Queen Mary College and ICL is now actively selling the DAP as an enhancement to its normal system products.
In the longer term, the DAP can be regarded as a new system component, a store with built in processing capability, which is likely to have far reaching effects on the evolution of systems engineering practice such as for example, the efficient implementation of distributed systems.

ARITHMETIC DOMINATED COMPUTATIONS	
Meteorology	13 x IBM 360/195
Magneto Hydrodynamics	14 x IBM 360/91
Structures	6 x IBM 360/195
Simulations	10 x CDC 7600

DECISION DOMINATED COMPUTATIONS	
Table Look Up	3 x CRAY 1
Pattern Matching	300 x IBM 360/195
Operations Research	1200 x IBM 370/145

TABLE 1: 64 x 64 DAP performance estimates on selected problems.

8.4 Man/machine interaction by speech

Our fourth major activity is concerned with Man/Machine Interaction by Speech. A clearly recognisable human habit in the communication of information is the preference for the "conversational" mode. It can be regarded as a behavioural adaptation to the imperfections of human communications since each individual message can be supplemented on request by repetition or clarification.
To carry on a conversation each participant must be able to reply quickly, before the last speaker has forgotten what he said and why. The CAFS system permits such rapid interaction using a keyboard and video display for man/machine communication. Now that we have a machine which can respond fast enough to be a credible conversational partner the ultimate objective is man/machine communication by speech to permit conversational working in the full sense of the word. This is a most difficult problem. The process of human speech communication by natural language is not fully understood and, certainly could not be reproduced by a machine. Our objective is to develop techniques which will enable practical speech communication with an information system to be effected retaining the major advantages of using speech, its ease of use, and efficiency as a means of information transfer.
It is perhaps useful to consider the requirements for a speech input/output system. It should be based upon the use of an ordinary telephone to avoid expensive terminal equipment and to make potential access to a very large number of users. It should incorporate standard digital technology which, on account of its intrinsically high speed compared with the information rate of speech, can be multiplexed to achieve low cost per channel. The use of digital techniques brings the usual advantages of reliability, repeatability and maintainability to what has traditionally been the province of analogue techniques.
The recognition device should be adaptive so that it compensates for peculiarities of the speaker and the individual telephone. In addition to the vocal adaptation the overall system should be designed to simplify the recognition problem by taking advantage of context. At each stage of successful communication in a conversation, the possible repertoire for the next communication is often known to be restricted and there is every reason to take advantage of this fact. By such means it is possible to match a simple machine to a human user capable of great subtlety without excessively annoying the user. Indeed, it is possible to a limited extent to make the machine detect whether a user is experienced, or casual and untrained and structure the interaction accordingly.
Speech output is an easier task for a machine than speech input and is likely to be of commercial significance sooner. We now have ready for exploitation a speech synthesiser, a powerful technique for speech output that can be implemented either in multi-channel or single-channel form. This development has great potential and will enable computer based information services (interactive and non-interactive) to give direct spoken information to the public. For example in a directory enquiry system, about 10 seconds of operation connect time is spent in relaying the telephone number to the enquirer. The use of a speech output device for doing this job would save an estimated £2 million a year in the UK.
Our research activities in the speech interaction field have inevitably caused us to be more aware of the intrinsic nature of information as a by-product of human life. This has been most valuable and will help us to develop techniques for handling the more complex input/output that will be a system requirement of the future.

9. Summary and conclusions

1. A typical information engine will be conceived as a subsystem to a natural human information system. This needs a little explanation. What do we mean by system and subsystem? In conventional usage we think of a computer as a system and its peripherals as subsystems. We used to think we could make a computer any way we liked but a peripheral device must be constrained to plug-in to the computer. We can now recognise that the computer itself is a subsystem in this sense, whose existence can be justified only if it is consciously designed to serve people whose behaviour is unnegotiable. All this will necessitate a new attitude of constructive humility to be adopted by information engineers.
2. We must exploit the order which users instinctively impose on their information and respect the disorder which arises from the fact that human affairs are not totally predictable. We are already well practiced in exploiting order since the design and use of high level languages is essentially directed to this end. However, we can expect to gain much advantage by providing the means for exploiting order at the lowest practicable level in our information system so that they can offer advantages for much of the system software as well as for the ultimate user. In present practice, disorder is not respected and is too often

regarded incorrectly as a failure of overall system design.

3. Technological advance permits such a system to be designed but does not guide how to do it. Some relevant techniques have been demonstrated in ICL Research.
4. Profound changes in information system practice are inevitable as a consequence of 1, 2 and 3.
5. However, the timing of such changes is difficult to predict since large scale events are controlled by a commercial stick/slip mechanism. This arises from the fact that all large scale decisions are quite properly made to maximise return on investment. The total situation evolves by the accumulation of understanding of objectives together with ripening technology. When such reasons for changes are less than decisive no changes occur at all since the right business decision is to obtain some more return from existing investment. When eventually the reasons become decisive a band-wagon effect occurs so that the changes occur more quickly than might be expected. In my judgement the "slip" is likely to occur in the next decade.
6. Genuinely modular system design will be increasingly practised and will lead to defacto standard functional specifications for modules.
7. A typical module will comprise a combination of hardware possibly including active storage modules derived from the Distributed Array Processor and software whose overall functional specification will be clearly understood by the user in ordinary human terms.
8. The user will be able to control the deployment of such modules to match his evolving requirements.

References

Variable Computer System

1. DENNIS, J.B. and VAN HORN, E.C. (1966). Programming Semantics for Multiprogrammed Computations, *CACM,* **9** No. 3.
2. ENGLAND, D. (1974). Capability Concept mechanism and Structure in System 250, IRIA Int. Workshop, Protection in Operating Systems Aug. 1974, 63-82.
3. EVANS. O.V.D. and MAY, J. (1975). The VCS System — An Overview of its Operational Aspects, VCS.20 RADC, ICL, July 1975.
4. FABRY, R.S. (1968). Preliminary Description of a Supervisor for a machine oriented round Capabilities, ICR Quart. Rep.18, ICR, U of Chicago Sec. 1B.
5. FABRY, R.S. (1971). List Structured Addressing, PH.D Th., U of Chicago.
6. ILIFFE, J.K. and JODEIT, J.G. (1962). A Dynamic Storage Allocation Scheme, *Comput* **J,5,** 200-209.
7. ILIFFE, J.K. (1968). Basic Machine Principles, MacDonald/ American Elsevier.
8. ILIFFE, J.K. (1969). Basic Machine Language, TR. 1021 ACTP Final Report. RADC, ICL 30. Sept.1969.
9. NEEDHAM, R.M. (1972). Protection Systems and Protection Implementations, Proc.AFIPS FJCC, **41,** AFIPS Press.
10. NEEDHAM, R.M. and WALKER, R.D.M. (1977). The Cambridge CAP Computer and its Protection System, SOSP6.
11. SALTZER, J.H. and SCHROEDER, M.D. (1975). The Protection of Information in Computer Systems, *Proc. IEEE,* **63,** 9.
12. "Variable Computer System", ACTP Final Report, TR.1157 RADC, ICL. Sept. 1974.

Content Addressable File Store

1. CODD, E.F. (1970). A Relational Model of Data for Large Shared Data Banks, *Comm ACM,* **13,**6.
2. COULOURIS, G.F., EVANS, J.M., MITCHELL, R.W. (1972). Towards Content Addressing in Data Bases, *Computer J,* **15,** 2.
3. LIN, C.S., SMITH, D.C.P. and SMITH, J.M. (1976). The Design of a Rotating Associative Memory for Relational Database Applications, *Transactions on Database Systems,* **1,** 1, 53-65.
4. MITCHELL, R.W. (1976). Content Addressable File Store, Online Database Technology Conference, April 1976.
5. OZKARAHAN, E.A., SCHUSTER, S.A., SMITH, K.C. (1975). RAP — An Associative Processor for Data Base Management, AFIPS National Computer Conference, **44,** 379-387.
6. SU, S.Y.W. and LIPOVSKI, G.J. (1975). CASSM: A Cellular System for Very Large Data Bases, Proceedings of the ACM International Conference on Very Large Data Bases, Sept. 1975, Framingham, Mass., 456-472.

Distributed Array Processor

1. FLANDERS, P.M., HUNT, D.J., REDDAWAY, S.F., PARKINSON, D. (1977). Efficient High Speed Computing with the Distributed Array Processor, in *High Speed Computer and Algorithm Organisation,* Academic Press, 113-128.
2. PARKINSON D. (1977). An Introduction to Array Processors, Systems International, Nov. 1977.

Man/Machine Interaction by Speech

1. ADDIS, T.R. (1972). Human Behaviour in an Interactive Environment using a Simple Spoken Word Recogniser, *International Journal of Man/Machine Studies,* **4,** 255-284.
2. ADDIS, T.R. (1978). Human Factors in Automatic Speech Recognition, RADC, ICL. Technical Note TN.78/1.
3. UNDERWOOD, M.J., ADDIS, T.R. and BOSTON, D.W. (1972). The Evaluation of Certain Parameters for the Automatic Recognition of Spoken Words, Machine Perception of Patterns and Pictures, Inst. of Physics Conference Series, **13,** 117-125.
4. UNDERWOOD, M.J. and MARTIN, M.J. (1976). A Multi-channel Formant Synthesiser for Computer Speech Response, Proceedings of the Inst. of Acoustics, Autumn Conference 1976 2-19-1.
5. UNDERWOOD, M.J. (1977). Machines that Understand Speech, *The Radio and Electronic Engineer,* **47,** 8/9, 368-376.

Notes for Contributors

The purpose of this Journal will be to publish original papers in any field of computing. Papers submitted may be research articles, review articles, exploratory articles or articles of general interest to readers of the Journal. The preferred languages of the Journal will be the congress languages of IFIP although papers in other languages will not be precluded.

Manuscripts should be submitted in triplicate to either Dr. D.S. Henderson or Prof. M. H. Williams at

Rhodes University
Grahamstown 6140
South Africa

Form of manuscript

Manuscripts should be in double-space typing on one side only of sheets of A4 size with wide margins. The original ribbon copy of the typed manuscript should be submitted. Authors should write concisely.

The first page should include the article title (which should be brief), the author's name, and the affiliation and address. Each paper must be accompanied by a summary of less than 200 words which will be printed immediately below the title at the beginning of the paper, together with an appropriate key word list and a list of relevant Computing Review categories.

Tables and figures

Illustrations and tables should not be included in the text, although the author should indicate the desired location of each in the printed text. Tables should be typed on separate sheets and should be numbered consecutively and titled.

Illustrations should also be supplied on separate sheets, and each should be clearly identified on the back in pencil with the Author's name and figure number. Original line drawings (not photoprints) should be submitted and should include all relevant details. Drawings, etc., should be submitted and should include all relevant details. Drawings, etc., should be about twice the final size required and lettering must be clear and "open" and sufficiently large to permit the necessary reduction of size in block-making.

Where photographs are submitted, glossy bromide prints are required. If words or numbers are to appear on a photograph, two prints should be sent, the lettering being clearly indicated on one print only. Computer programs or output should be given on clear original printouts and preferably not on lined paper so that they can be reproduced photographically.

Figure legends should be typed on a separate sheet and placed at the end of the manuscript.

Symbols

Mathematical and other symbols may be either handwritten or typewritten. Greek letters and unusual symbols should be identified in the margin. Distinction should be made between capital and lower case letters between the letter O and zero; between the letter l, the number one and prime; between K and kappa.

References

References should be listed at the end of the manuscript in alphabetical order of author's name, and cited in the text by number in square brackets. Journal references should be arranged thus:

1. ASHCROFT, E. and MANNA, Z. (1972). The Translation of 'GOTO' Programs to 'WHILE' Programs, in *Proceedings of IFIP Congress 71,* North-Holland, Amsterdam, 250-255.
2. BÖHM, C. and JACOPINI, G. (1966). Flow Diagrams, Turing Machines and Languages with only Two Formation Rules, *Comm. ACM,* **9**, 366-371.
3. GINSBURG, S. (1966). *Mathematical Theory of Context-free Languages,* McGraw Hill, New York.

Proofs and reprints

Galley proofs will be sent to the author to ensure that the papers have been correctly set up in type and not for the addition of new material or amendment of texts. Excessive alterations may have to be disallowed or the cost charged against the author. Corrected galley proofs, together with the original typescript, must be returned to the editor within three days to minimize the risk of the author's contribution having to be held over to a later issue.

Fifty reprints of each article will be supplied free of charge. Additional copies may be purchased on a reprint order form which will accompany the proofs.

Only original papers will be accepted, and copyright in published papers will be vested in the publisher.

Letters

A section of "Letters to the Editor" (each limited to about 500 words) will provide a forum for discussion of recent problems.

Hierdie notas is ook in Afrikaans verkrygbaar.

Quaestiones Informaticae

Part 2 of the proceedings of the first South African Computer Symposium on Research in Theory, Software, Hardware, organised by The Research Symposium Organising Committee of The Computer Society of South Africa. 4 & 5 September 1979, Pretoria.

Contents/Inhoud