QUAESTIONES INFORMATICAE

Vol. 1 No. 1

June, 1979

Quaestiones Informaticae

An official publication of the Computer Society of South Africa
'n Amptelike tydskrif van die Rekenaarvereeniging 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.

Editorial Advisory Board

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.

A Hardware-Based Real-Time Operating System

M. G. Rodd

Dept. of Electrical Engineering, University of the Witwatersrand, Johannesburg, South Africa

Abstract

The efficient use of multiprogrammed industrial control computers is largely a function of the relationship between hardware and software. A shift in this relationship is desirable, since multiprogrammed computers typically spend a large proportion of computing time in handling their own organization. This situation is compounded in many time-critical industrial process-control applications.

This paper proposes that a possible solution lies in the adoption of a hardware-based real-time operating system. The system consists of a microcontroller working in close relationship with a conventional minicomputer. To retain a high degree of flexibility, the microcontroller makes use of microprogrammable, bipolar, bit-slice microprocessor elements. In essense, the unit executes the principal functions of a real-time operating system, acts as a pre-processor for all incoming requests, and ensures a high rate of task switching.

The system has been applied in a series of real-time experimental configurations. These were controlled successively by the conventional, software-implemented approach, and by the proposed system. The respective performances were evaluated. The new strategy is shown to result in a better and more economical industrial controller.

1. Introduction

The introduction of the electronic computer has had an immense impact on society. In almost every aspect of life, this product of man's power of innovation is exerting an influence. One particular area in which the computer's ability to perform high-speed, repetitive tasks is used to an ever-increasing extent, is the industrial sector. In an era of inevitable expansion of industrial capacity, industrialists look to the computer to aid them in achieving their goals of higher production and more constructive use of labour. The demands made on the controlling computers have necessarily increased, from the execution of relatively simple sequential tasks, to an extent where total control of complex processes is invested in the computer.

As the demands increase, the capabilities of the computers must increase likewise. While dramatic technological advances have resulted in faster, more reliable, machines there is, seemingly, a limit to what a single computer is capable of performing. The solutions offered have been many — multiply the complexity of the computer; add hardware; develop more sophisticated software; introduce additional, parallel processors.

The majority of such complex systems have been produced by means of ad-hoc design procedures, based on experience and intuition. This paper shows how a simplistic analysis of the performance of a single processor can reveal the vital criteria to be considered when defining a computer structure for a specific industrial application. Based on this analysis, a system is proposed which provides for a highly efficient industrial controller [2]. The heart of the structure is a microcontroller, executing control over a minicomputer and acting both as a partial preprocessor and as a hardwired operating system.

2.1 Organizational Problems of Process-Control Computers

In broad terms, a computer applied in a real-time process-control situation must be capable of performing the following functions:

(i) The scheduling of a variety of tasks in such a way as to effect overall control of the process. ("Task" is taken to imply a sequence of computer instructions which perform a predetermined system function [1]. Each task will have a certain time which it takes to complete and will, normally, require execution at certain prescribed intervals.

(ii) The initiation of a specific task, indicated (or "requested") by any of the following occurrences:
   (a) A signal derived from the plant.
   (b) A request from another task currently in execution.
   (c) A signal from one of the computer system's own external or internal peripherals (e.g. an event-timer or a real-time clock).

(iii) The temporary suspension of a task currently in execution, following a request for a task with a higher priority in the predetermined, pre-emptive priority hierarchy.

Reviewing the above demands on such a system, one is struck by the fact that the problem which confronts the computer systems designer is by no means unique. The situation is typical of many systems, having limited resources while being subjected to demands for service at varying intervals.

Typical of such a system is a telephone exchange. An Exchange might have thousands of subscribers, but can link only relatively few of them at any one time. The requests for connections occur at some statistically evaluatable rate, and the lengths of calls (or holding times) may also be statistically predicted. Such systems have been studied for many decades, and the results obtained can be of great value to the systems analyst in many other fields. In the case under discussion in this paper, it is a single-server model which is appropriate, i.e., only one request can be honoured at any one time, with all other requests occurring during that time being placed in a waiting queue [3].

The assumptions made are as follows:

(i) That the incoming requests are characterised by a Poisson distribution, with an average request rate of K (per time-span).

(ii) That the distribution of task lengths may be characterised by a negative exponential function of the form

$$f(t) = \frac{1}{h} e^{-t/h}$$

where h is the mean execution time of the tasks, and t is the time variable.

It may be shown (4), that the resulting mean queue length, L, is given

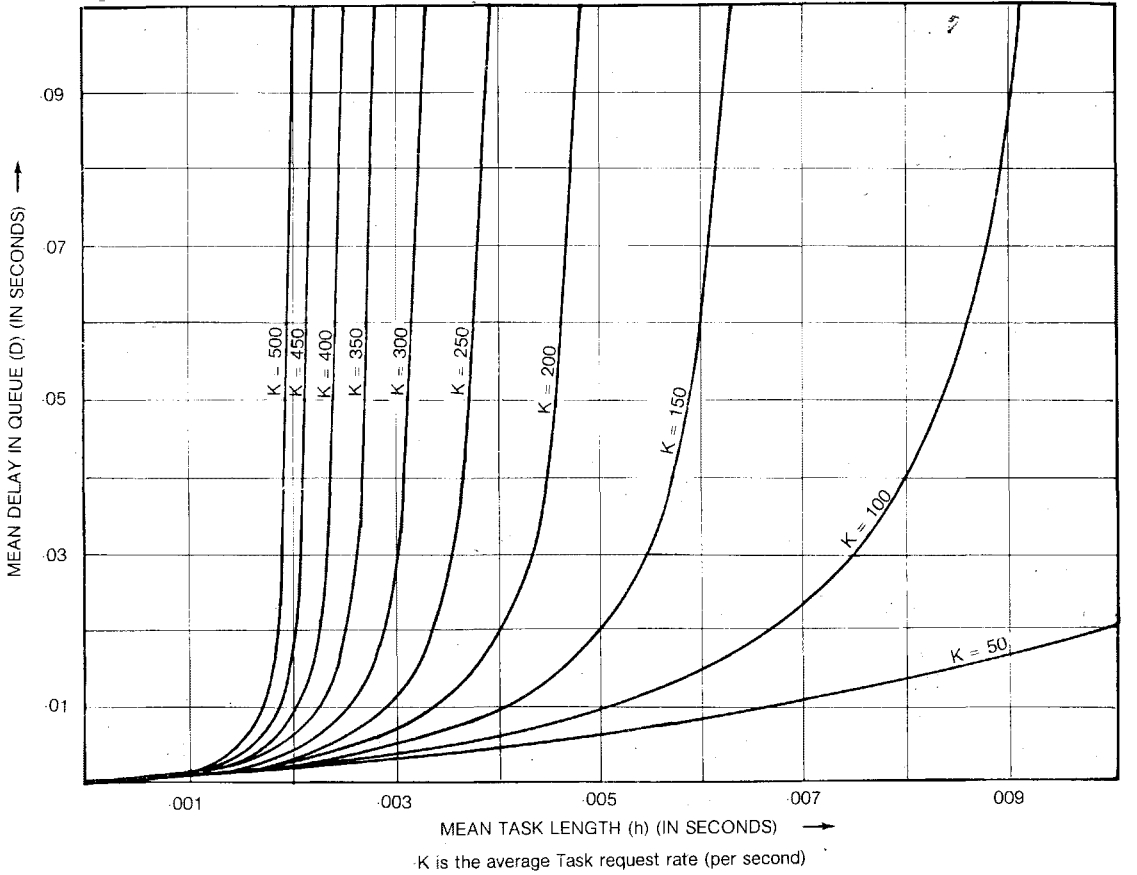
$$L = \frac{Kh}{1-Kh} \qquad (1)$$

and the mean delay, D, of tasks in the queue is given by

$$D = \frac{h}{1-Kh} \qquad (2)$$

Fig. 1 plots expression (2). In the most basic terms, this gives the key to the performance of an industrial control computer. The model is obviously highly simplified, as problems such as priority are not considered. (Ref [4] shows how these can be included). The analysis does, however, provide a firm basis for viewing the requirements of a computer system performing such functions.

This paper was presented at the SACAC symposium on Real-time Software for Industrial Applications in Pretoria on 29-30 November, 1978.



2.2 Organizational Strategies

The criterion of acceptability of any control system is that it must be capable of performing the required function! In terms of Fig. 1, this may be interpreted as requiring that the execution of every task, when requested, should occur within a permissible limit (any delay would, of course, arise from the queue which may develop).

If the characteristics of a process under review are such that non-permissible delays will occur if a conventional single computer is used, how can the system be structured to produce an acceptable situation? Fig. 1 indicates the way. This shows that, to reduce the average delay time experienced for a specified number of tasks, only two parameters can be varied. These are:

(i) The average task request rate; and
(ii) The average task execution time.

To achieve a reduction in either or both of these parameters, a variety of techniques may be employed. Some of the more obvious are outlined below.

(a) **Reduction of the Average Request Rate**

Two solutions are immediately apparent here. Firstly, pre-processing may be employed to sift through demands from the system (i.e., requests for attention, alarms, etc.), and to select those which are absolutely vital for sending on to the main computer. Secondly, processing power may be distributed among a hierarchy of processors. Thus, the request rate (and therefore the load) to any one machine will be reduced.

(b) **Reduction of the Average Execution Time**

The obvious indication is that actual run times of tasks should be kept to an absolute minimum. This implies, firstly, that the processor should be made as fast as possible, and secondly, that programs should be optimally coded for minimum run-time. There are, however, two important factors which should also be considered here. Firstly, it is common knowledge that any multi-programmed computer spends much of its available computing time in executing internal operating system functions and virtually "trying to decide what to do next". Such operating system functions can be considered as tasks and should therefore be kept to a minimum. Also, the question of swop-time (the time taken to switch execution from one task to another) must be taken into account. Since the bringing into execution of any task will require a task-swop, this time-length may be included in the average task execution time. Thus, reduction of the task-swop-time will effectively reduce average execution time.

3.1 A Possible Solution

The factors discussed above can be considered collectively, and a multi-faceted approach adopted. It is not intended to attempt a totally new architectural approach to computer design. The area of immediate interest lies in the modification of existing minicomputer systems to produce more capable and efficient systems.

It is proposed that the solution lies in the supervision of the functions of a minicomputer by means of a separate, but closely-linked, hardware unit. The functions of the unit are:

(i) To execute the primary executive functions of a real-time operating system.
(ii) To pre-process all incoming requests from the process and from the peripherals, as well as those generated by tasks currently in execution in the computer.
(iii) To supervise task-swopping within the minicomputer.

As well as meeting these requirements, it is essential that:

(a) The unit may be interfaced to the minicomputer **without** modifications to the existing hardware.
(b) It should be economically effective.
(c) Although a hardware unit, the structure must retain a high degree of flexibility.

The block diagram of a suitable structure is shown in Fig. 2. The hardware unit comprises a hardwired operating system together with a local memory, an interrupt controller, and interfacing to the computer. The interfacing to the minicomputer utilizes the computer's input/output bus, and requires no change in the minicomputer's existing hardware.

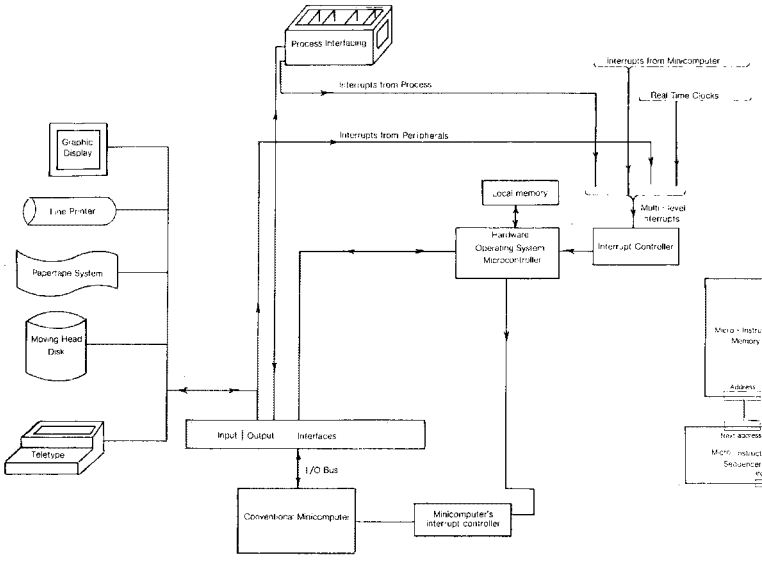


FIGURE 2 : SYSTEM BLOCK DIAGRAM

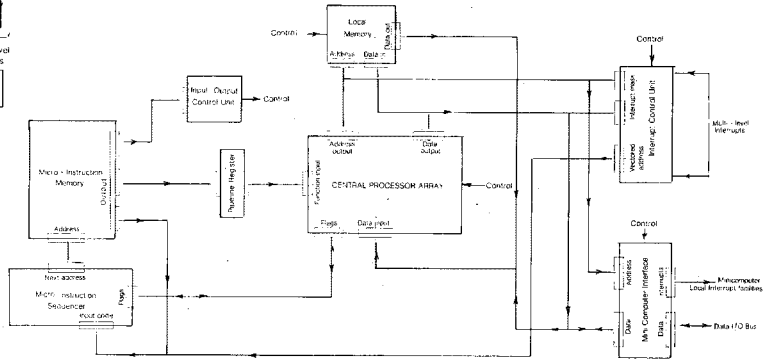


FIGURE 3 : MICROCONTROLLER BLOCK DIAGRAM

PARAMETER	COMPUTER I WITH MICRO-CONTROLLER	COMPUTER I WITH KERNEL IN-CORE REAL-TIME OPERATING SYSTEM	COMPUTER II WITH IN-CORE, REAL-TIME OPERATING SYSTEM (16 Bit)	COMPUTER III WITH REAL-TIME OPERATING SYSTEM (32 Bit)
Time to recognise the presence of an interrupt	1,8	7,2	10,6	6
Time to accept an interrupt and initiate the task requested. (I)	171	408,6	261,5	2106
Time to effect a task-swop. Note (i)	165,6	408,6	200	2100
Time to initiate a task called by another task. Note (i)	171	471,6	200	2300
Time to react to an interrupt generated by a peripheral. Note (i)	1,8	7,2	10,6	6
Time to initiate an input/output operation. (i)	171	408,6	±300 Note (iii)	1800
Time to terminate an input/output operation.	171	408,6	±300 Note (iii)	1700
Longest non-interruptable time. Note (ii)	7,7	495	Not available	97
Core required to implement a basic operating system (in actual locations.)	76	508	2805	±10K

3.2 A Practical Implementation

From the required system functions, it is clear that the hardware unit (referred to below as a "microcontroller"), is in essence, a high-speed computer with somewhat limited and specialized functions. The high speed is essential, since it must be capable of rapid reactions to incoming requests, and must be able to control the minicomputer as effectively as possible. It was estimated [2], that the microcontroller should be one order of magnitude faster than the minicomputer in processing speed. This indicated a desired cycle-time of the order of 150-200 nano-seconds, which immediately ruled out conventional mircroprocessors. The need for flexibility pointed to a microprogrammable unit. To meet both these requirements, bipolar bit-slice microprocessor elements were used (see Fig. 3.). The central processor array is a 16-bit wide system, for convenient data exchange with the host computer. The micro-instruction word is essentially vertical in structure, with a 24-bit width. Two pages of micro-instruction memory, each with 512 words, are included. Information relevant to each task (sometimes referred to as a Process Control Block) is held in a local, high-speed bipolar memory, consisting of 1 K x 16 bits RAM. The interrupt control unit presently comprises 64 levels of interrupt, fully vectored, with programmable masking. The minicomputer interface controls data flow, and routes signals to the minicomputer's interrupt system. All communications with the minicomputer are effected via this interface, which also controls the necessary synchronization during data transfer. To effect data transfer, approximately 70 locations are required within the minicomputer's memory, to act as a buffer store and interrupt catcher.

3.3 Operation

The real-time operating system is effected by microprogrammes. At the present time, its functions are limited to those of the executive of a conventional, in-core, real-time operating system. As such it may be regarded as a Kernel operating system. Only limited file-handling has been implemented. The primary functions of the operating system are:

(i) Scheduling of multiple tasks. The tasks may be either real-time dependent and priority-based, or non-time-critical background tasks. A limited degree of dynamic rescheduling is permitted.
(ii) Control of multiple, vectored interrupts.
(iii) Control of inter-task communication and synchronization, including buffer-passing, as well as communication between tasks and the executive.
(iv) Supervision of all input/output operations.

In simple terms, the microcontroller determines which task should be executed at any one time. It examines all incoming interrupts, and uses a single-queue, priority-based, pre-emptive scheduling algorithm to determine which tasks should be swopped-in or-out. Input/output requirements in the minicomputer are dealt with by regarding their driver routines as tasks, on the same basis as user tasks.

3.4 System Evaluation

The system outlined above has been implemented in conjunction with a minicomputer, applied in a variety of real-time control situations [2]. The results obtained have been compared with results measured using conventional techniques. The same minicomputer system was used, under the control of an in-core, real-time operating system. In addition, comparisons were made with other minicomputer systems operating under their current real-time operating systems. Some of the results obtained are shown in Table 1.

TAKE IN TABLE 1

TABLE 1: **Comparison of Some Key Real-Time Operating System** Parameters

NOTES: A: All times are in microseconds.
B: (i) Assumes the requested task is in memory, and has a higher priority than the current task.
(ii) Only considers tasks which form part of the operating system, and excludes any user tasks.
(iii) Dependent on the peripherals used.

A detailed comparative study was also made of the Utilization Factor (UF), a factor which indicates what proportion of the total available computing time is actually used for valuable work (i.e. the processing of user tasks). In a typical, fast on-line control application, it was found that a UF of < 0,08 could be expected, i.e. less than 8% of total available computing time was actually devoted to the execution of user tasks. The rest of the time was taken up by scheduling, task-swopping, interrupt handling, etc. Under the supervision of the microcontroller, a UF of up to 0,7 (i.e. 70%) could be achieved in the identical situation.

4. Conclusion

It has been described how a simple statistical analysis of a process-control computer system can indicate ways of ensuring efficient organization. It is clear from the analysis that the provision of bigger and better facilities will not necessarily result in improved systems. In order to improve the system and to provide for a high degree of utilization of the facilities available, there are certain clear-cut factors which must be considered. These factors indicate that an unconventional, integral hardware/software approach to the management of industrial control computers would ensure highly efficient operation. Such possibilities become feasible in view of current technological developments, specifically with the introduction of ultra high-speed microcomputer components.

Indeed, the work described in this paper points to an important avenue, in which much research effort should be concentrated; that of the supervision of large, powerful mainframe computers by simple, low-cost controllers. It is clear that any efficient system implies efficient use of resources. Thus, simple, logical tasks such as scheduling and interrupt handling are not necessarily the forte of powerful, 'number-crunching' computers, and the delegation of such tasks to these machines is a misuse of potential power. The solution lies in surrounding the mainframe machines with small, limited-capability computers (i.e. microprocessors), which will supervise and assist the central machines in using their potential to the utmost. To quote Maurice Wilkes, "A computer operating system may be regarded as — and indeed is — a control system. There would thus appear to be scope for implementing an operating system by means of a number of interconnected micro-processors dedicated to the purpose." [5]

Acknowledgement

The author wishes to express his appreciation to the University of Cape Town for assistance in the production of this paper.

References

1. BARRON, D.W. (1971). *Computer Operating Systems,* Chapman and Hall, London.
2. RODD, M.G. (1976). *Organization of Industrial Control Computers,* Ph.D Thesis, University of Cape Town.
3. RUBIN, M. and HALLER, C.E. (1966). *Communication Switching Systems,* Rheinhold Publishing Corp., New York., 246-271.
4. SYSKI, R. (1960). *Congestion Theory in Telephone Systems,* Oliver and Boyd, London. 298-340.
5. WILKES, M.V. (1977). The Application of Microprocessors to Main Frame Design, in *Proc. IERI Conference on "Computer Systems & Technology",* Brighton, U.K.

Real-Time Interactive Multiprogramming

A. D. Heher

NEERI, CSIR, Pretoria, South Africa*

Abstract

This paper is concerned with the problems of interactive computing when multiple real-time tasks are involved. The interactive facilities which can be provided are described and some of the important software engineering concepts which should be incorporated in modern programming systems are discussed. Memory management is a crucial activity in interactive systems and a new memory management technique called Software Virtual Memory Management (SVMM) has been developed to allow the interactive facilities of monoprogrammed systems to be extended and improved in a multiprogramming environment. This management technique has been incorporated in an interpreter-oriented system called VIPER (Virtual Interactive Process Executive for Real-time control) and experience with this system is reported here. The paper concludes with a brief analysis of the performance of VIPER which shows that the SVMM technique can be used to improve the performance of interactive real-time systems constructed with the aid of an interpreter to the extent where they can be competitive with compiler-oriented systems in certain applications.

1. Introduction

In terms of both processing power and ease of use, a significant gap exists in real-time operating systems between stand-alone BASIC-type systems and disc-based compiler-oriented executives. There are many applications where it is desirable to utilize the interactive facilities of a language like BASIC while retaining the multiprogramming capabilities of the compiler-based executives. This paper discusses the problem of constructing such a real-time interactive multiprogramming system and describes a system which has been implemented to demonstrate the concepts involved.

The objective of bridging this gap is to permit more complex programming tasks to be undertaken by application-oriented staff by providing them with simple but powerful software tools. The success of this approach has been reported by a number of workers [2-5, 7-8, 19-22, 24]. Despite their limitations and language defects, interactive systems like BASIC, supplemented by a range of high-level real time functions, have proved useful in a variety of applications. To encourage the programming task to be undertaken by those who understand the problem, the task of programming should be simplified in every possible way. This permits users to get into the task 'without undue effort spent in learning computerese' [3], and also 'to improve software reliability by reducing the opportunity for error' [7].

The simplicity and security of the software system which is used are important factors in the production of reliable software [16], particularly if more complex multiprogramming tasks are to be undertaken. In constructing a system which supports multiple concurrent tasks, it is not sufficient to merely extend the facilities of monoprogrammed systems; radically different software techniques must be applied in order to maintain and improve upon the reliability achieved by these simple systems.

There are five aspects to which particular attention must be paid in real-time systems, viz

1. Data structure and access.
2. Protection of code and data areas.
3. Synchronization and control of concurrency.
4. Structured programming.
5. Program and system documentation.

Before dealing with these aspects in Section 3, three introductory topics are discussed in Section 2. The discussion in this paper is strongly related to an interactive system which has been constructed. An overview of this system is therefore given first (2.1) to enable the discussion which follows to be placed in context, followed by a discussion of memory management in interactive systems (2.2). The third topic deals with the concept of interaction itself; what is meant by interactive and the interactive facilities which can, and should, be provided (2.3).

2.1 An Overview of Viper

VIPER is an interpretive system which evolved from an earlier monoprogrammed real-time BASIC called PROSIC [9, 10]. PROSIC in turn was a development from the original Varian BASIC. [6]. VIPER is coded in Varian Assembler and, like BASIC, is a stand-alone system containing all its own operating system functions. It has been run on a Varian 620 and on a microprogrammable microprocessor system called MIKROV [26]. The memory-resident portion of VIPER is 13 K words (16 bit) in size.

VIPER permits independent, named segments of code and data to be executed and manipulated concurrently. Each of the code segments is a self-contained procedure which is similar to a stand-alone BASIC program in many respects. The procedures (= code segments) are created and manipulated interactively from an input device.More than one keyboard can be active at once as VIPER has a multi-user, multi-terminal capability, as well as multiprogramming facilities. Other tasks in the system can also run concurrently while program development is proceeding. At any given time an input device is associated with a particular procedure and all commands and statements are executed within the scope of that procedure. The association of a device and procedure can be changed with simple commands. Fig. 1 illustrates some of these interactive operations. Figs. 2, 3, 4 and 5 illustrate other types of VIPER statements which are described in the sections that follow.

The segments of code are moved to and from a bulk storage device under control of a memory manager. At any given time some, or all, of the modules of a task may be swapped out of memory to make room for other modules of the same, or some other, task. If a module which is not in local memory is referenced, directly or indirectly, by any command or statement, it is swapped-in under control of the executive. The logical space required by a set of modules can therefore be larger than the physical space. This gives VIPER virtual-memory properties.

All statements have the same syntax, irrespective of whether they are executed as commands or as program statements. In other words, the command and programming languages are synonymous. This duality not only simplifies the user interface but also results in the protection and data manipulation facilities being applied equally to the command and programming languages. Statements are differentiated from commands by the presence or absence of a line number. One of the most important properties of VIPER is that interactive operations, including the execu-

This paper was presented at the SACAC Symposium on Real-time Software for Industrial Applications in Pretoria on 29-30 November, 1978.

*The author is now with AECI Limited in the Consulting Engineers Department.

tion of commands and the addition of statements, can continue while a procedure is executing. Operations of this type are illustrated in Fig. 1. The provision of interactive debugging operations on executing real-time tasks is not merely a convenient feature — it is a powerful tool for the testing and debugging of real-time software. Hoare [16] has noted that this commissioning of software can be "the most tiresome, expensive and unpredictable phase" and any tool which can simplify and shorten this phase can make an important contribution towards the goal of producing more economical and reliable software.

Shared data areas are an important resource in a real-time environment and require protection from inadvertent or illegal modification if the system is to be secure. Shared data segments are provided in VIPER which are referenced and defined in a manner analogous to that of named COMMON blocks in FORTRAN IV, with the significant difference that the segments can be created and deleted dynamically like files, protected like files and moved to and from input/output devices. Figs. 2 and 3 illustrate some of the commands and statements available for the manipulation of data elements. Shared and local data facilities are described in more detail in Section 3.1.

2.2 Memory Management

Interactive programming systems require that any statement in a task can be changed, deleted or added in some sort of incremental compilation mode i.e. the entire task or procedure need not be recompiled and link-loaded. A good interactive system should also support interaction during the execution of the task with monitoring and debugging facilities that do not require the suspension of the task before they are activated. In PROSIC, the forerunner of VIPER, it was demonstrated that even more general interactive facilities can be provided in a mono-programmed system [9, 10], which it would be desirable to extend to the multi-tasking environment.

A number of interactive systems with either multiprogramming or multi-user capabilities have been constructed [4, 19, 23, 29]. However, they all suffer from the following shortcomings:

1. A lack of independent named procedures and subroutines which are essential for a structured programming approach.
2. Poor shared data facilities and a lack of protection for any of the facilities provided.
3. Restricted interactive facilities. None of these systems, nor any system known to the author, permits interactive operations to be applied to executing tasks.

These shortcomings can all be traced to a single problem: memory management. The implementation of interactive facilities requires that the code defining a task and its associated data areas be expanded and contracted as the interaction proceeds. In a multiprogrammed system the difficulty occurs in the attempt to allow multiple tasks or procedures to undergo this dynamic change in size and structure simultaneously. The addition of a multi-user capability further complicates the memory management task, as does the need for flexible access to shared data areas.

These considerations led to the development of a new memory management technique. This management system is implemented entirely in software but has many of the characteristics of a system using hardware virtual memory management, and for this reason the technique used has been called 'Software Virtual Memory Management' (SVMM). As a result of the software implementation, a resident operating system nucleus must exist in a fixed memory partition and only the remaining memory is available for virtual storage operations.

The term 'virtual memory' has two connotations in the context of this paper: the first is related to the usual concept of addressing a logical space which is larger than the physical space; and the second is related to the security of, and access to, both tasks and data structures which are operated upon as if they were located in a file system. Both executable (and executing) tasks and data structures are afforded protection in a hierarchy of security levels. The user therefore creates, modifies and executes tasks as if he were working on a set of files which may in fact be memory-resident; and conversely, he operates within a task as if all tasks and data structures were memory-resident when in fact they may be resident on some external device. This file-system analogy is an extension of the usual concept of virtual memory in that it is associated with the reverse mapping of memory onto a mass-storage device, as opposed to the mapping of mass-storage onto memory, which is the property of the extended logical space.

The memory management structures have been described elsewhere [13] and as they are fairly complex, space precludes a discussion of their operation in this paper.

2.3 Interactive Operations

The term "interactive" has acquired a variety of meanings in computer applications. Two basic divisions which can be identified are:

1. Interactive program development.
2. Interactive dialogue in an applications environment (e.g. data-base management and information systems).

The second category is important in process control applications as part of the interface between the computer system and the process engineers and operators, but it is the first category which is of primary concern to this paper. Even the term "interactive program development" is not well-defined — it is used by some authors to refer to computing services of the time-sharing type and by others to incremental compilation and direct execution, such as is possible with BASIC. Another context in which the term "interactive" is used, is in mini-computer operating systems where the user drives the system directly from a keyboard to edit, compile, load and test programs in a rapid development cycle. The term "interactive" arises from the fact that on modern disc-based operating systems these operations can be performed in one or two minutes as opposed to 15 to 30 minutes on older magnetic-tape or paper-tape oriented operating systems. Although a great improvement on past systems, this type of operation is not considered interactive in the context of this paper.

The interactive facilities which are provided in VIPER fall into four interrelated and overlapping categories:

1. Symbolic debugging of programs on-line and in real-time.
2. Monitoring of on-line real-time programs; examination of plant variables and perturbation of outputs.
3. Creation of new programs and editing of old programs.
4. Testing the modules of a task as they are developed. (Top-down design and step-wise refinement.)

Only two functions need to be implemented to enable these facilities to be provided:

1. The ability to add a statement to (or delete it from) a procedure at any time, whether it is executing or dormant.
2. The unification of the command and programming languages.

These functions unify the language elements, the debugging and monitoring commands and the file manipulation commands into a single coherent set with a common syntax, and enable the interactive mode of operation to remain active on executing tasks. The operation of a process can therefore be dynamically monitored and symbolically debugged by means of the same command and programming language used to write the program. In PROSIC, the monoprogrammed predecessor of VIPER, the essential simplicity and naturalness of this on-line real-time debugging and monitoring facility proved to be an extremely powerful tool which was readily accepted by the process-oriented users. To enable these facilities to be extended to VIPER, however, the properties of SVMM are essential as this level of interaction could not otherwise be supported in a multi-user multi-tasking environment.

The testing and debugging phase can be further simplified if they are combined with the coding phase by use of an interactive software development system. The interaction permits software modules to be tested as they are written, or as soon afterwards as possible and allows iterations in

the software development cycle with the rapid testing of previously developed modules as additional modules are added. Interactive testing and debugging is particularly important in real-time systems where a complex set of programs co-operate to perform a given task in response to real-time events. If a task needs to be stopped or taken off-line before 'test' or 'debugging' functions can be included, the commissioning task is made considerably more difficult and timeconsuming.

Wilkes [23] made some pertinent comments in this connection: "There has, to my mind, been too little interest in devising efficient methods for locating the errors that do get introduced. Most debugging procedures in current use are crude and depend on examination by the programmer of a static picture of his program when it has stopped. Methods of obtaining a trace of what was happening during the running of a program have been successfully used in the past and I suggest that the time has come to re-examine these methods with the object of developing them into serious tools that can be used by the software engineer."

3.1 Data Structure and Access

The organization and protection of the data elements in a programming system are recognised to be crucial factors in the development of reliable software [15]. In VIPER there are three main categories of data types, viz local variables, shared data, and variables passed as parameters in a procedure call. Protection facilities are applied to all three categories and they are all accessed by means of similar mechanisms. The dominant characteristic of the data element is that most of them can be created dynamically.

The links between segments which are required to reference items in another segment are established dynamically on the first reference to the variable (incurring some overhead), but , once established, the links are maintained until released by either the user or the system. After establishing the links, subsequent references are therefore performed efficiently with a minimal overhead.

1. Local variables: The only local variable types provided so far in VIPER are simple and array floating-point variables. Provision has, however, been made for additional types including string, bit and integer variables and these could be included within the SVMM framework without difficulty. Local array variables can be defined dynamically or statically. Within a subprocedure, for example, a variable size area can even be allocated in accordance with a passed parameter and then deallocated (released) before exiting from the procedure. Fig. 3 shows a schematic outline of this type of operation. Read or write access to an array can also be specified.

2. Shared data elements: Shared data elements are grouped together in named segments which can be manipulated by the SVMM routines. The syntactic structures for creating and referencing these elements are similar to those used for named COMMON blocks in FORTRAN, as illustrated in Fig. 2. Both simple and array variable elements within a particular segment can be individually read- or write-protected. A segment can also be password-protected to prohibit access to all but password holders. Data segments could be swapped like procedure segments, but this has not been implemented in VIPER. They can, however, be moved to and from bulk storage devices under user control. All data segments also have a semaphore associated with them which can be used for synchronization, as discussed in Section 3.3.

3. Parameter passing: Parameters can be passed from one procedure to another using a normal FORTRAN-like CALL-SUB-RETURN sequence. The invoked subprocedure need not be contiguous to or owned by the calling procedure, however, and may not even be resident in memory when it is invoked. Parameter types are matched and must agree and default access states are established where appropriate. If a constant appears in the actual parameter list, for example, the corresponding formal parameter state is set to read-only. Specific access states of actuals are also copied through to the formulas. The detection of illegal mappings (mismatched types) is performed at the CALL-SUB set-up time while access violations are checked on each reference to a formal parameter. Although there is a certain overhead involved in this detailed verification of parameter passing, the checking is considered essential in view of the fact that this interface is one of the most troublesome and error-prone areas in the programming. The SVMM structures minimise this overhead and operations involving formal parameters take only 4% longer than the equivalent operation using local variables only. More dramatic, is a factor of four improvement in VIPER compared with PROSIC, despite the extra protection functions.

3.2 Protection

The types of protection facilities which can be applied to data elements in a real-time system were discussed in the preceding section. Of equal inportance in an interactive system is the protection of code segments. Protection functions are provided in many operating systems but these often apply only to bulk storage resident files. In VIPER, the protection facilities are applied to executable code (and data) segments and remain in force on active tasks. The ability of users to modify procedures, access data areas or execute tasks can therefore be controlled dynamically. The application of file-system-like protection facilities to active segments in the system is a unique property of SVMM.

The protection mechanisms have two goals — the first is to provide facilities which are easy to use, and the second is to ensure that they are impossible to circumvent. These two goals conflict at times so that, in practice, a modicum of effort must be expended to achieve the highest level of protection; on the other hand, good protection facilities are always applied by default without any explicit user action.

The basic mechanism for access protection is a capability list which is associated with each segment; the primary function of this is to control a set of well-defined states in which a procedure can operate. These states provide protection for the user against himself as well as against 'pirates'. A password is used to control the commands which can be used to manipulate (indirectly) entries in these capability lists. Before any input is accepted from a user at a keyboard he must LOGON with an appropriate password.

A password is not necessarily associated only with a particular user. Its primary function is to logically partition tasks into sets of co-operating procedures. The set of procedures and the associated data elements which control a particular section of a plant, for example, can be associated with a particular password, while the modules of an operator interface could be given another. In this context the LOGON command identifies a logical subset of procedures which the user wishes to access. It also serves the usual protection function, however, in that no modifications can be made to any of the procedures if the appropriate password is not specified. (It may still be possible to examine and execute the procedures if this has been permitted by their 'owner'.)

The control of access to code segments illustrates the type of functions which can be provided by means of these protection facilities. A procedure can exist in one of four primary states, together with several substates.

Change: In this mode any alteration can be made to a program, even if the program is executing. It is the basic mode used to edit programs and with a little care it can also be used as a debugging mode in that permanent changes to the program can be made immediately.

Debug: This mode possesses a restricted set of the CHANGE mode access rights. The procedure can be listed, variables examined and breakpoints and statements inserted, but no existing statements can be deleted or modified. Statement execution frequency counts can also be invoked in this mode.

Monitor: This mode permits a procedure to be examined using commands such as PRINT and LIST, but no statements can be added or changed. This restriction ensures that nothing can be done which interferes with the execution f a procedure and this mode can therefore be made freely available to process staff. A substate of the mode can be invoked, however, to limit access to password holders.

Execute: This mode has three substates; free execute; password execute; and no access. The latter category enables the execution of a procedure to be prohibited until one of the other substates is established with a specific command.

The modes CHANGE, DEBUG and MONITOR are entered with commands of the form CHANGE ‹procedure name› while substates are specified by commands with the form ACCESS (‹procedure name›) = ‹attribute› . Only the appropriate passholders can enter the CHANGE and DEBUG modes or change the access attributes.

3.3 Synchronization

The semaphore is the basic building block for the synchronization of processes and the control of access to shared data. It is, however, an awkward element to use in real-time programming, for several reasons:

1. If a lock (wait) operation is encountered in the program text it is not immediately clear whether or not it is an entry to a critical section, in which case it should be followed by a free (signal) operation further on.
2. If it is the entry to a critical section it may not be immediately obvious from the text what the shared variables are.
3. It is difficult to check whether all critical sections are properly protected by a semaphore.
4. It is difficult to check for the possibility of deadlock.

For these reasons other language constructs, such as the "REGION" construct [1] and the "MONITOR" [14] have been proposed. Hoare's monitor concept has been noted to be the most general and secure structure, but it would appear to be more suitable for operating system structure than for an application oriented software system like VIPER. Reviewing the synchronization and protection requirements of such systems, the "REGION" construct was selected as the one which appeared most natural for use with the shared data segments so extensively used in VIPER. This operates as follows:

Given a shared data area which is declared with a statement

COMMON ‹com name› , ‹data list›

a critical region where mutually exclusive operations are required is defined by:

REGION ‹com name›
‹critical region statements›
END REGION ‹com name›

Two or more procedures declaring an area in this way are guaranteed to be mutually exclusive in the critical region. The REGION statement sets a semaphore associated with the data area and can only proceed to execute the critical region statements if the semaphore is not already locked. If the semaphore is locked, the procedure is suspended and waits for the semaphore to be cleared (unlocked) by an END REGION statement. In addition to the REGION construct, primitive operations are provided for the direct manipulation of semaphores. These simple but powerful facilities assist in the modular decomposition of tasks into separate and independent sub-tasks which are safer and which are much simpler to code and debug.

3.4 Structured Programming

"I take structured programming to be a term of art signifying a style of programming in which the flow of control is determined by procedure calls and by statements of the type IF . . . THEN . . . ELSE . . ., rather than by the indiscriminate use of GOTO statements. Further, it is usually advocated that the program should be written in a top-down manner. These recommendations, it is claimed, lead to a disciplined method of programming with the following advantages:

1. The program, being modular in nature, is easy to understand and check.
2. There is a possibility of proving it correct.
3. It is easier to maintain and modify."

M.V. WILKES [28]

The term "structured programming" has acquired a variety of meanings, but Wilkes' concise statement captures the essential properties of this programming discipline. The development of structured programming techniques is a current topic of research and a wide variety of control structures have been proposed and discussed. Because of this fluidity only the simplest and most widely used structures were used in VIPER. There are two aspects of structured programming which are of particular relevance to VIPER; viz. modularity and structures.

3.4.1. Modularity

In VIPER each named code module, which may be either a procedure or a subroutine, exists as a separate segment which can be independently moved to and from bulk storage devices. One of the goals of structured programming is to break up a task into modules, each of which is no more than one to two pages in size (30 to 70 lines of code). In SVMM, therefore, a well-structured program is naturally divided into blocks a few hundred words in size, each of which represents a natural "page" that can be swapped to and from a bulk storage device. This 1:1 correspondence between pages and segments is in marked contrast with hardware virtual memory mapping devices where the page boundaries are randomly scattered over the procedures constituting a task.

One of the recommended practices associated with the art of structured programming is the independent testing of the individual modules of a task as they are written. Some sophisticated software tools have been developed for this type of operation, particularly for cases where top-down design or stepwise refinement strategies are used. VIPER makes no specific provision for this design procedure but the ease with which modules can be tested individually, together with the flexible data structures which simplify the generation and linking of test data, enables this practice to be carried out with the aid of the standard interactive facilities. Of more importance than a formal design procedure (which is possibly of relevance only to large software problems which would most probably not be coded in VIPER in any event) is the informal flexibility of being able to test and examine the operation of a procedure in a variety of ways before it is finally integrated into an overall task.

3.4.2 Structures

The control structures incorporated in VIPER are as follows:
IF - THEN - ELSE -ENDIF
FOR - NEXT
DO WHILE - END DO
CASE - ENDCASE
GOTO

This restricted set of relatively simple structures was chosen as they were considered adequate for the type of software likely to be written in VIPER. Examples of the use of these structures are given in Fig. 5. Despite the simplicity of the structures they have a markedly beneficial

effect on both the clarity and the ease of understanding of the programs.

The simple GOTO was retained in VIPER as it has quite clearly been shown [18] that it is sometimes required, even in well-structured programs, to avoid awkward and clumsy constructions. An interesting observation arose, however, from the case study discussed in section 4. In the translation of approximately 1 300 lines of FORTRAN code into VIPER not a single GOTO was required, whereas the FORTRAN code contained nearly 100 of them. This observation indicates that the control structures chosen are adequate for the relatively simple logic structures that generally occur in process control work.

One of the most important aspects of structured programming in an interpretive system is that it can be used to automatically perform the indenting that provides the invaluable visual aid to program structure. An example illustrating this facility is given in Fig. 5. The manual insertion of indenting is a tiresome and frequently overlooked chore which is especially difficult when programs are changed or updated. Furthermore, real programs are subject to a steady flow of changes and improvements over their lifetimes [16], so this problem is not just a development phenomenon. In VIPER the automatic indenting is coupled with a proof of the structural correctness of the program. This proof is not only an assurance that the program is correctly structured, but is also a useful teaching aid in that it gently prompts the user to use the correct constructions, pointing out the cause of the error and where it occurs. With this interactive assistance users unfamiliar with structured programming can rapidly learn the rules.

In addition to the control structure indenting there is another aspect of program layout which is of importance in real-time programming. Programs which execute cyclically nearly always require an initialization section where control loop variables and items in common areas are given initial values. The static initialization performed by FORTRAN-type data statements is only a partial solution, as the initialization requirements can encompass all programming functions, including input/output operations and computations-based or process variables. This is achieved in VIPER by providing a statement START which indicates the end of the initialization section and the start of the repetitively executed code. The initialization code is intended to distinguish it from the body of the program. Examples of this facility can be seen in Figs. 3 and 5.

3.5 Documentation

Interactive systems like BASIC or VIPER are frequently used for experimental or investigatory work, an environment where the maintenance of good documentation is as difficult as it is important. Interpretive systems also frequently suffer from the disadvantage of not being able to use source text layout to improve program visibility and understanding because of the back-listing or decompilation this is performed to recreate text. Special effort must therefore be made to assist and encourage the documentation of interpretive programs. Both program and system documentation functions are important, as discussed below.

3.5.1. Language structure

The most important factor in the production of clear, well-documented software is the language structure itself. No quantity of comments can overcome basic defects in the language. There are four important properties of language structure, viz:

Structured language. This is one of the most important aids to program documentation and is absolutely essential to enable interpretive systems to back-list (decompile) a program in an intelligible format. This aspect was commented on in Section 3.4 and an example of the VIPER facilities is given in Fig. 5. There is a strong case for all interpretive systems to use a structured language, for the sake of documentation, if nothing else.

Variable and procedure naming conventions. The restrictions in BASIC (a letter and a digit for simple variables and a letter only for array variables) are quite unnecessary, as an extension of PROSIC has shown [10]. In VIPER, all names, including variables, data areas and procedure names, can be up to 16 characters in length. These longer names are an invaluable aid to clear documentation and reduce the need for trivial comments to explain the meaning of variables.

Syntactical structures. The syntax of the language and operating system commands can contribute to the readability of programs. The structured programming keywords such as CASE, DOWHILE and the synchronization function REGION are examples of syntactical structures with clear and unambiguous meanings. Other functions of importance are operations such as RUN and WAIT. Some examples illustrating the syntax of these statements in VIPER are given in Fig. 4. The action which is required is immediately apparent without reference to manuals to determine the meaning of obscure parameters.

Conciseness. FORTRAN and BASIC suffer from a lack of conciseness which results in program modules being physically larger than necessary. As the ease with which a program module can be understood is related to its size, there is an incentive to allow more compact representations. (Conciseness, in the dictionary sense of "short and clear", is not to be confused with the sententious contraction of a language like APL.) One simple but successful aid is to allow multiple assignments on a line. This is effective because simple assignments are the most common statements in typical programs [17]. As the assignment statement does not affect the program flow, this conciseness does not detract from program clarity. It is the control structures IF-FOR-CASE and the like which determine the flow and these are pivots on which the understanding of a program hinges; contracting the "straight-line" code enhances the lucidity of the control structures. The comment conventions adopted in VIPER, which are discussed in the next paragraph, also contribute to maintaining the conciseness of programs.

3.5.2 Comment facilities

All languages make provision for comments in one form or another, but the actual syntactical forms used are of crucial importance [16, 25] The ease with which comments can be inserted, and their readability once inserted, are important factors in determining the extent to which the facilities will be used by programmers. End-of-line comments are especially recommended as they are easily inserted, are directly associated with a line of code, and can be made highly visible. In VIPER end-of-line comments are right justified in the back-listing operation, as shown in Fig. 5. This achieves the required visibility while maintaining conciseness.

3.5.3. System documentation

Typical real-time programming tasks are made up of a number of independent modules which operate on one or more data bases. In maintaining and operating these systems it is important to understand the relationships between the various modules of the task, and to know which modules call others (the hierarchical relationship) and which modules access particular data areas. The relationships amongst modules is of importance because the interface amongst them is known to be one of the most troublesome and error-prone in real-time programming.

A number of documentation systems which produce the required information from an off-line analysis of the source program listing have been reported on in the literature. (It is assumed that the only precise and up-to-date source of internal documentation for most software is the programs themselves.) However, the programs are typically large (10 000 — 15 000 statements) which illustrates the complexity of producing the information from source listings. In interpretive systems, and in the SVMM structures of VIPER in particular, the information is readily

available within the various symbol tables, and system documentation information on the overall structure of the task can easily be produced on-line. This is a particular advantage in an interactive system in that the information that is produced represents the actual state of the system at that time. Dynamic information on structures that vary with time can also be produced.

4. Performance

Interpretive systems have a reputation for poor performance which is not always entirely justified, as the data below shows. (This reputation would appear to have arisen, at least in part, from the notoriety of some early BASICs which interpreted source code directly.) Data published by a number of authors indicate that in typical applications, interpretive tasks are approximately five times slower than the equivalent compiled tasks. Similar results have been obtained from comparisons between VIPER and FORTRAN programs. This direct comparison does not always give the complete picture, however, as illustrated by the case study below. Before these results are discussed, the relationship between VIPER and other interpretive systems is briefly mentioned to place VIPER in perspective.

1. Comparison with interpreters. A variety of simple benchmark programs have been run to compare the performance of VIPER with that of its monoprogrammed predecessor PROSIC, as well as with that of a number of other BASIC or BASIC-like systems (10, 11). From the measurements that have been made it has been observed that the performance of VIPER is similar to that of both PROSIC and other BASICs when considering small benchmarks (5-20 statements). In larger programs (100-200 statements) the performance of VIPER is better than that of PROSIC by a factor of two or three and in the 900-1 000 statement category VIPER can achieve a performance improvement of up to 10 to 1. This effect and results of one particular benchmark are tabulated in Tables 1 and 2. This large program superiority of VIPER results directly from the modular SVMM structures and their efficient linking mechanisms. They avoid the increase in execution time in accordance with program size, which is a characteristic of many BASIC systems.

2. Comparison with compiled code (a case study). A FORTRAN-based process control package had been developed for an experimental process control investigation. This had ddc and supervisory control modules in addition to scanning, logging and alarm functions,. The system ran on a Hewlett Packard 21MX computer under control of the RTE II Executive in 32K words of memory. In the configuration used, 19 primary real-time tasks (15 of which ran cyclically) executed in a single foreground partition, resulting in numerous disc swapping operations. Semaphores were used extensively for synchronization and to control access to shared data which consisted of a single (unprotected) global common area. The semaphores facilitated the modular decomposition of tasks, but contributed to the high swapping rate. The FORTRAN programs were recoded into VIPER, resulting in 35 SVMM segments (27 code and 8 data), 16 of which execute repetitively. Table 3 summarises the statistics of the FORTRAN and VIPER programs. Two results of this case study are of interest:

Memory requirements. The VIPER programs require approximately one third of the memory space required by the FORTRAN programs and all the repetively executed modules can be memory-resident in a 32K word machine.

Execution time. Averaged over 60s, the repetitive tasks consume 1,2 s of CPU time in the FORTRAN system versus 7,9 s in VIPER; a ratio of 6,6 to 1. When the time spent swapping is taken into account, however, the real-time foreground partition of the RTE system is busy for 9,0 s in 60 s compared with the 7,9 s in VIPER. In terms of real-time task throughput, the interpretive VIPER system is therefore slightly faster than its FORTRAN-based counterpart.

Ignoring all differences between the two computers used the results indicate that VIPER, running on a microprogrammed microprocessor emulator [26] is capable of substantially the same throughput of real-time tasks as a real-time executive which executes in-line compiled code with swapping. The Hewlett-Packard RTE system could also support concurrent tasks in the resident and background partitions and it is not claimed that VIPER is equivalent to a system like the HP RTE in computational power. What is claimed, is that given a set of real-time tasks such as those encountered in the case study, an interpretive system can have much the same performance as a compiler oriented system and could be used in many applications where much larger and more complex operating systems had to be used previously.

5. Conclusion

It has been shown that interpretive systems are capable of significantly better performance when the techniques of software virtual memory management are employed. As the structures required can be implemented entirely in software, the system is machine-independent and can be implemented on any average mini- or microcomputer. The software system which has been constructed has a uniform command and programming language, and this simplifies operation and enables outstanding debugging facilities to be provided in an interactive, multi-user mode of operation. Incorporated into the system are excellent protection mechanisms which are easy to use but permit a number of users co-operatively to share a secure data base in a real-time environment. Despite the ease of use and the extensive protection mechanisms, the performance of the system is better than that of similar interpreter-based real-time systems and is competitive with compiler-oriented systems in some applications.

Acknowledgements

The support by the National Electrical Engineering Research Institute of the work reported in this paper is gratefully acknowledged. I would also like to express my thanks to my supervisor, Prof. H. L. Nattrass, of the University of Natal, for his guidance and advice. Many members of the NEERI staff also assisted with the work and I would in particular like to note the efforts of our software librarian Mrs. N. D. Thomson for many months of patient text editing, Mr. H. Frommann for work on the Varian Cross Assembler, Mr. J.P. Los for constructing the MIKROV computer, Mr. P. Grift for maintaining (and improving) our old Varians and Mr. P. Hussey for writing many of the FORTRAN programs in the case study and for translating some of these programs into VIPER.

References

1. BRINCH HANSEN P. (1973). *Operating System Principles,* Prentice Hall, Englewood Clifs.
2. CLAGGETT, E. (1977). Interpreters versus compilers for on-line microcomputers, *Control Engineering,* August 1977, 24-27.
3. DIEHL, W. (1976). Software for industrial computer control — a review, *1st IFAC/IFIP symposium on software for computer control,* Tallin, USSR, 279-285.
4. DIEHL, W. and SANDERS, D. (1975). Real-time BASIC used in a distributed network, *IFAC/IFIP workshop on real-time programming,* Boston, 159-163.
5. GAINES, B.R. and FACEY, P.V. (1975). Some experience in interactive system development and application, *Proc IEEE,* **63,** 894-911.
6. GOUWS, J. H. (1973). *User's notes on the process BASIC interpreter,* Internal report Ea-41, National Electrical Engineering Research Institute, Pretoria.
7. GRIEM, P. D. (1975). On the principle of unique definition, in *Proc AFIPS NCC,* **44,** Anaheim, 265 - 270.
8. HAASE, V. M. (1976). Evaluation of BASIC as a programming language for real-time systems, *IFAC/IFIP workshop on real-time programming,* Roquencourt, 331-339.
9. HEHER, A.D. (1976). Some features of a real-time BASIC executive, *Software Practice and Experience,* **6,** 387-391.
10. HEHER, A.D. (1976). PROSIC: A real-time BASIC executive, in *Proceedings of SACAC symposium on minicomputers and microprocessors,* Durban, 56-65.
11. HEHER, A.D. (1977). A real-time interactive software system for process modelling, optimization and control, *5th IFAC/IFIP digital computer applications to process control.* The Hague, 647-653.
12. HEHER, A.D. (1977). Refinery Computer Control Project Software organization and CAMAC drivers, CSIR Special Report ELEK 130, CSIR, Pretoria.
13. HEHER, A.D. (1978). Software virtual memory management in a real-time interactive software system, to appear in *IEEE Trans. on Software Engineering.*
14. HOARE, C.A.R. (1974). MONITORS: An operating system structuring system, *Comm ACM,* **17** *549-557.*
15. HOARE, C.A.R. (1975). Data reliability, *Proceedings of IEEE international conference on reliable software,* New York, 528-533.
16. HOARE, C.A.R. (1975). Hints on programming language design, *Real-time system implementation languages,* U.S. Army Research and Development Group (Europe) Tech. Report ERO-2-75, vol. 2, 505 - 534.
17. KNUTH, D.E. (1971). An empirical study of FORTRAN programs, *Software Practice and Experience,* **1,** 105-133.
18. KNUTH, D.E. (1974). Structured programming with GOTO statements, *ACM Computing Surveys,* **6,** 261-301.
19. KOPETZ, H. and CRIGHTON, E.R. (1976). Ergonomics and security of real-time programming, *IFAC/IFIP Workshop on real-time programming,* Roquencourt, 331-339.
20. LARSON, S.L.G. (1974). Adapting BASIC for process control — meeting the needs of a conversational control system, *Advances in Instrumentation,* **29,** 512-517.
21. LAURANCE, N. (1975). Structured programming in a real-time environment, *IFAC/IFIP workshop on real-time programming,* Boston, 49-58.
22. MAPLES, M.D. and FISCHER, E.R. (1977). High-level language applications at Lawrence Livermore Laboratory, Univ. of California, U.S.A., *Microprocessors,* **1,** 312-316.
23. PERSEUS COMPUTING AND AUTOMATION (1976). MULTEX-BASIC manual WU-000168-01.
24. SCHOEFFLER J.D. and KEYES M.A. (1976). Hierarchical language processing, *1st IFAC/IFIP symposium on software for computer control,* Tallin, USSR, 263-272.
25. SCOWEN, R.S. and WICHMANN, B.A. (1974). The definition of comments in programming languages, *Software Practice and Experience,* **4,** 181-188.
26. VAN AARD, J.L. (1977). Microprocessor emulation of a Varian minicomputer, report NIDR 77/03, CSIR, Pretoria.
27. VAN MEURS, J. and CARDOZO, E.L. (1977). Interfacing the user, *Software Practice and Experience,* **7,** 85-93.
28. WILKES, M.V. (1976). Software engineering and structured programming, *IEEE Trans. on Software Engineering,* **SE-2,** 274-276.
29. WILKINS, A.G. (1976). Swepspeed-1 user guide, Central Electricity Generating Board, London, SSD/SW/75/N16.

Figure 1: A Short Example Illustrating Some Interactive Operations

INPUT (Output not shown)	INPUT DEVICE ASSOCIATION	COMMENT
LOGON USER1	MASTER	USER1 = password (echo of input is suppressed during LOGON) Creates a procedure called USER1.
PROC ABC	USER 1	Create a procedure called ABC and associate input device with it. ABC has default password USER1.
10 . . .	ABC	Enter statements into ABC (in any order)
20 . . .	.	
.		
.		
.		
PROC XYZ	ABC	Create XYZ (Input now associated with XYZ)
100	XYZ	Enter statements
50 . . .	.	Enter statements
.	.	
.		
CHANGE ABC	XYZ	Return to make a change to ABC (only permitted to password holder USER1)
200 . . .	ABC	Change a statement in ABC
RUN XYZ EVERY 5 SECS	ABC	Set XYZ to execute periodically
RUN (ABC)	ABC	Execute ABC-(ABC) optional (defaulted) because of input device association
PRINT X	ABC	Examine variable X in ABC while ABC is running
MONITOR XYZ	XYZ	Monitor operation of XYZ (restricted rights)
PRINT Y	XYZ	Examine variable Y in XYZ while XYZ is running
DEBUG ABC	XYZ	Enter restricted mode (No changes to existing statement permitted)
100 PRINT X	ABC	Insert statement to examine X at line 100 (ABC still executing)
CHANGE (ABC)	ABC	Move to CHANGE mode to permit alterations
110 . . .	ABC	Make a change
PRINT X	ABC	Examine X now
STOP (ABC)	ABC	Terminate execution immediately
TURNOFF XYZ	ABC	Remove XYZ from time list
SAVE	ABC	Save copy of ABC on external device
SAVE XYZ	ABC	Save XYZ
LOGOFF	MASTER	End of session, return to Master. Deletes procedure USER1.

Figure 2. Some Examples of Shared Data Manipulation

CONSOLE INPUT	COMMENT
LOGON USER1	Password USER1 will be associated with all COMMONS created
COMMON SIZES, N1, N2	Construct a data area (this is a command).
ACCESS (SIZES) = WRITEA	Permit write operation
N1 = 100; N2 = 120	Initialise this COMMON
PROC XYZ	Create procedure XYZ
10 COMMON SIZES, N1, N2	Link to SIZES to pick up N1 and N2 Default access is read only.
20 COMMON COM1, A(N1), B(N2)	Set up variable size data area
30 COMMON COM2	No data area, semaphore only.
40 ACCESS(A)=READA+WRITEA;	A: read and write;
ACCESS (B)=0	B: not used here (no access)
.	
.	
.	
100 REGION COM1	Start of a critical region (Mutually exclusive access to COM1)
.	
.	
.	
160 A(. . .) = . . .	Perform some operation on A
.	
.	
.	
180 SAVE COM1	Save current values on bulk storage device
200 ENDREGION COM1	End of critical region
210 FREE COM2	Unlock semaphore associated with COM2 (see ABC line 100 below)
.	
.	
.	
250 DELETE COM1	Delete COM1 and allocate new size
280 COMMON COM1, A(N1*2)	
.	
.	
.	
.	
PROC ABC	Create procedure ABC
10 COMMON COM2	Declare semaphore (no data)
.	
.	
.	
100 LOCK COM2	ABC will suspend until FREE COM1 in line 210 of XYZ
.	
.	
.	
LOGOFF	

Figure 3: Data Manipulation Example

```
SUBROUTINE SUB A(N,X)
   M = 20                          Dimension according to a variable value
   DIM A(M)                        Static allocation
   A(1)=10; A(2)=27; A(3)=. . .    Assign some values
   ACCESS (A) = READA              Protect by setting access to read only
START                              End of initialization section
DIM B(N), C(N, 3*N)                Dynamic allocation of array space
.
.
.
.                                  Perform operations with arrays
DIM B(0), C(0)                     De-allocate before exit
RETURN
```

Figure 4: Examples of Run and Wait Statement Syntax

```
RUN STARTUP                        Run once
RUN LOOP. CONTROL EVERY 5 SECS
                                   Execute repetitively
LOGH= 1,5                          Times can be fractional
RUN LOOP. LOG EVERY LOGH HOURS IN 2O*LOGH MINS
                                   Repetitive after initial delay
NEXTSHIFT=8*INT(HOURS+8)/8)
                                   Next shift time (HOUR=current time)
RUN SHIFT. REPORT EVERY 8 HOURS AT NEXTSHIFT: 0
                                   Run at shift changeover
RUN DAILY.REPORT AT 08:00:30 EVERY 24 HOURS
                                   Run every day at 30 secs after 8 a.m.
WAIT  1.3 SECS                     Floating point values and expressions
WAIT X*Y/Z MINS                    permitted, HOURS is also a qualifier
```

FIGURE 5: AN EXAMPLE OF THE STRUCTURING OPERATIONS

```
#PROC STRUCTURE.TEST
#1 PROC
#10 PRINT "SIMPLE STRUCTURE TEST"
#20 START END OF INITIALISATION CODE
#100 FOR I=1 TO 7 MAIN LOOP
#110 PRINT I,
#11_20 IF I<3 BINARY IF ON ITS OWN FOR VISIBILITY
#130 THEN PRINT "  I<3", THEN,ELSE AND UNARY IF CAN ONLY
#140 ELSE PRINT "  I>=3", BE FOLLOWED BY A NON-CONTROL
#150 IF I=4 PRINT "  I=4", STM ON THE SAME LINE
#160 ENDIF
#200 IF I>=5
#210 THEN THE FOLLOWING CONTROL STM MUST BE ON A NEW LINE
#220 FOR J=1 TO 4
#230 CASE J=1 OUTER CASE INDEX=J
#240 PRINT "  CASE J=1",
#250 CASE I=6 NESTED CASE INDEX=I
#260 PRINT "  CASE I=6",
#270 CASE I=7
#280 PRINT "  CASE I=7",
#290 ENDCASE I END OF INNER CASE
#300 CASE J>2 AND I>6 COMPOUND CASE CONDITION, INDEX=J
#310 PRINT "  CASE J>2 AND I>6",
#320 ENDCASE END OF OUTER CASE
ERROR 3 IN LINE  320 OF STRUCTURE.TEST     (Example of syntax error handling.)
  320 ENDCASE
#320 ENDCASE J END OF OUTER CASE
#330 NEXT J
#340 ENDIF
#350 PRINT "  ◆"
#400 NEXT I END OF LOOP, LINE NO LINKS FOR STM
#999 END PROC NAME ADDED BY SYSTEM
#LIST
VIPER  REV  A7  12/04/78  20:53:01.7  18/04/78

#    1 PROCEDURE STRUCTURE.TEST
   10   PRINT "SIMPLE STRUCTURE TEST"
   20 START STRUCTURE.TEST                                  END OF INITIALISATION CODE
  100 FOR I=1 TO 7                                                        MAIN LOOP
  110   PRINT I,
  120   IF I<3                                        BINARY IF ON ITS OWN FOR VISIBILITY
  130     THEN  PRINT "  I<3",                          THEN,ELSE AND UNARY IF CAN ONLY
  140     ELSE  PRINT "  I>=3",                           BE FOLLOWED BY A NON-CONTROL
  150       IF I=4 PRINT "  I=4",                                 STM ON THE SAME LINE
  160   ENDIF
  200   IF I>=5
  210     THEN                    THE FOLLOWING CONTROL STM MUST BE ON A NEW LINE
  220       FOR J=1 TO 4
  230         CASE J=1                                             OUTER CASE INDEX=J
  240           PRINT "  CASE J=1",
  250           CASE I=6                                        NESTED CASE INDEX=I
  260             PRINT "  CASE I=6",
  270           CASE I=7
  280             PRINT "  CASE I=7",
  290           ENDCASE I                                          END OF INNER CASE
  300         CASE J>2 AND I>6                      COMPOUND CASE CONDITION, INDEX=J
  310           PRINT "  CASE J>2 AND I>6",
  320         ENDCASE J                                          END OF OUTER CASE
  330       NEXT J  220
  340   ENDIF
  350   PRINT "  ◆"
  400 NEXT I  100                                  END OF LOOP, LINE NO LINKS FOR STM
  999 END STRUCTURE.TEST                                 PROC NAME ADDED BY SYSTEM
RUN
#SIMPLE STRUCTURE TEST
1  I<3  ◆
2  I<3  ◆
3  I>=3  ◆
4  I>=3  I=4  ◆
5  I>=3  CASE J=1  ◆
6  I>=3  CASE J=1  CASE I=6  ◆
7  I>=3  CASE J=1  CASE I=7  CASE J>2 AND I>6  CASE J>2 AND I>6  ◆
RUN
#1  I<3  ◆
2  I<3  ◆
3  I>=3  ◆
4  I>=3  I=4  ◆
5  I>=3  CASE J=1  ◆
6  I>=3  CASE J=1  CASE I=6  ◆
7  I>=3  CASE J=1  CASE I=7  CASE J>2 AND I>6  CASE J>2 AND I>6  ◆
```

Table 1: Viper VS Prosic and HP Basic

Total number of statements [1]	Seconds to execute loop VIPER[2]	PROSIC[3]	HP BASIC[4]
7	4.4	4.4	3.0
50	4.4	8.5	6.6
100	4.4	12.8	10.6
200	4.4	21.3	18.7

```
100 FOR I = 1 TO 1 000
101 IF I < 500 THEN 104
102 X = I
103 GOTO 105
104 X = I
105 NEXT I
106 END
```

Notes:
(1) The additional statements are outside the loop.
(2) VIPER running on MIKROV [26]
(3) PROSIC on MIKROV [26]
(4) Hewlett Packard BASIC 20392A running on HP21MX

Table 2: A Benchmark

COMPUTER AND LANGUAGE	MILLISECS PER LOOP
Published Data (10)	
Data General 840 multi-user BASIC	4.5
DEC PDP 11/45 BASIC	3.2
DEC PDP 8E FOCAL	38.0
INTEL 8080 BASIC	75.0
INTEL 8080 compiled BASIC (Lawrence Livermore Laboratory)	22.0
VIPER running on MIKROV [26]	12.0
VIPER using floating point firmware and reverse Polish	4.2

```
10 REM SIMPLE BENCHMARK
15 REM *, /, -, +
20 REM
30 LET A = 1
40 LET B = RND(A)
50 LET C = A + B
60 LET A = A + 1
70 LET E= B/C
80 LET F = A*E
90 LET C = C-F
100 IF A = 1001 THEN 200
110 GOTO 50
200 PRINT "THE LOOP IS DONE"
210 END
```

Table 3: Case Study Program Statistics

	Module type	No. of modules	No. lines source code[1]	Code size words	Averages Lines module	Words line	Words module	Time busy per 60 secs CPU	Seconds Swap	Total
Fortran	A	15	1 178	32 276[2]	78.5	27.4	2 151	1.2	7.8	9.0
	B[4]	4	238	19 807	59.5	83.2	4 951			
	C	7		34 085						
	D	1	—	758	(memory resident global common areas)					
	E	10	(Disc files)							
Viper	A	16	638	12 501[3]	42.7	18.3	781	7.9	0	7.9
	B[4]	8	171	2 830	21.4	16.5	353			
	C	—	(not required because of interactive facilities)							
	D	8	—	390						
	E	3	123	2 799	41.0	22.8	933			

A — Repetitive tasks
B — Non-repetitive or infrequent
C — Monitoring and operator
D — Common data areas
E — Error messages

Notes
(1) Excluding comments
(2) Including non-reentrant library modules
(3) Including symbol table and SVMM overhead
(4) Type B functionally equivalent but not comparable line-for-line
(5) Hewlett Packard RTE FORTRAN 92060 - 16092 on 21MX.
(6) VIPER running on MIKROV (18)

Distributed Computer Systems — A Review

N. J. Peberdy

Dept. of Electrical Engineering, University of the Witwatersrand, Johannesburg, South Africa.

Abstract

The past five years have seen a dramatic changeabout in traditional hardware/software relationships: hardware costs have plummeted, and the size, environmental requirements and reliability of computing elements have altered drastically. It now becomes feasible to distribute a computing system, such that processors may be placed adjacent to the processes they control. These distributed computing modules operate in an essentially parallel mode, but are required to communicate in order to co-ordinate their activities. Reliable, secure communication systems must be established to ensure correct operation. Such systems are not only functions of the electrical hardware employed, but also of the software support provided. Of vital importance are the protocols selected, which define and detail an agreed procedure for the exchange of information.

This paper reviews the fundamental software considerations in the design of computer networks, with specific relevance for process-control applications. It discusses in detail, inter-connection strategies and protocols and briefly examines currently adopted schemes. The implications of fully decentralized system control are considered. Of particular concern is the question of the production of reliable, fault-tolerant, secure systems.

1. Introduction

Interest in multiple-processor systems has been generated by the quest for improved system performance. In the past the high cost of processors has limited investigation into such systems. However, due to the recent dramatic fall in hardware costs and processor costs in particular, distributed systems are becoming increasingly feasible. For example, one finds networks of computers being employed in large data-base systems to enable pooling of resources, or permit access to common data-bases. State-of-the-art mainframe computers typically consist of a number of processors, each dedicated to a particular task such as I/O, arithmetic processing, memory management, etc. In the process-control environment, the prospect of improved system performance has led to many so-called distributed control systems. In such a system the plant is partitioned into groups of concurrent tasks, each being controlled by a separate processor, with interconnection to enable co-ordination on a global basis.

The variety of ways in which computers may be interconnected has led to a problem of semantics. Terms such as distributed processing, distributed computers, networks, multiprocessors, etc., are being used in an almost random fashion to describe fundamentally different systems. This is because distributed computers involve new concepts and ideas which have yet to be clearly defined and resolved. For the purposes of this paper, we define a "distributed system" to be a multiplicity of computers that are physically and logically connected together and which co-ordinate their activities on a global basis under centralized or decentralized system control. The term "fully distributed computer" refers to a distributed system in which overall executive control is fully decentralized. The significance of these definitions will become apparent as the paper progresses.

We restrict our interest to those systems in which several processors are physically separated and connected together by data links. Further, we are restricting our area of application to those systems in which the various computer nodes co-ordinate so as to exercise overall control of some system which is itself complex and physically distributed.

2. Requirements of Real-Time Control

In real-time control systems, performance is measured more in terms of response-time than throughput. Events tend to occur in bursts which cause a large and immediate increase in processing activities. This occurs for example, when the plant moves towards a boundary condition. Failure to service a time-critical event within a specified time period could constitute a system failure.

Reliability is one of the most important criteria of plant control. This involves a number of aspects. The control system hardware and software should be correct, that is, error free. However, correctness is not a sufficient condition for reliability because of the finite probability of hardware malfunction. Reliability, therefore, also encompasses the concept of fault-tolerance, whereby the system continues to function in the presence of faults. In practice it is not possible to guarantee correctness particularly in the case of software. "Survivability" of the system involves confinement of initial errors, detection and diagnosis of failures, and finally recovery whereby further damage is prevented and the system is returned to a stable, consistent state.

During design of a plant control system, specifications are often changed, possibly due to changes in the plant itself. In addition, changes are invariably necessary during the life of a plant. The control system must be amenable to such changes. This calls for a highly modular structure both in hardware and software, so that changes in one module can be made without affecting other modules to any great extent.

Modularity involves another aspect. In uniprocessor control systems, each design tends to be different from others. This is especially true if one compares large and small systems. However, in a highly modular distributed system, large and small systems can be configured with the same basic design by varying the number of modules.

The final criterion is cost: any proposed system must be cost effective.

3. Uniprocessor Real-Time Control

Before the advent of low cost processors, plant control was (and in the majority of cases still is) exercised by a single computer. The computer performs all data acquisition, processing, and storage for logging purposes.

It also implements the control function for several processes, which must essentially execute in parallel. In order to perform these different tasks, the computer requires a large, complex executive to allocate CPU time to different tasks. Requests for service are typically asynchronous. If these requests are time critical then either polling at regular intervals is required, or interrupts must be permitted.

Operating systems are notoriously complex structures. The statement by Dijkstra that testing of software can only show the presence of bugs, not their absence, unfortunately sums up the situation. The inherent unreliability of this large software structure is further complicated by interrupts. Such a complex structure can have only very limited reliability.

At a more abstract level, the software designer constructs a virtual architecture which consists of a number of processes, each of which controls one or more plant processes. However, there is no structural correspondence of this virtual multi-purpose- architecture, to the actual uniprocessor architecture [40]. The mapping from the virtual to the real architecture must invariably introduce implicit relationships between processes which were not envisaged by the software engineer. Thus interaction may occur at random, causing very obscure errors.

This paper was presented at the SACAC symposium on Real-time Software for Industrial Applications in Pretoria on 29-30 November, 1978.

All-in-all, the multiprogrammed uniprocessor can have only limited reliability, regardless of speed or computational power. The solution must be found in other areas.

The reason for the complexity of the uniprocessor software is that in the past processors were expensive, and therefore had to be utilized to the full. However, hardware costs have plummeted whereas software costs are rising. It no longer makes sense to maximise processor efficiency. Instead, attempts should be focused on minimising and simplifying software, possibly at the expense of hardware.

4. Distributed Real-Time Control

Due to cost pressures and the requirement for greater reliability, multiple-processor systems have generated considerable attention in the process-control area. At this stage, the state-of-the-art is undeveloped and it will be many years before the potential advantages of distributed systems are fully exploited. However, even now, there are advantages in choosing such a system.

Perhaps the most typical distributed system consists of a number of small processors connected into an hierarchical network under centralized control. The plant is partitioned into a number of concurrent processes. Several processors are then dispersed around the plant, each controlling a process, or a group of processes in close physical proximity. The processors are connected into a network by cables to enable communication. These localized control processors then perform local data acquisition and control. The central computer is freed of the laborious task of data acquisition, and of much of the detailed control function. Its typical function would be to calculate set-points for the other nodes on the basis of selected data transmitted to it from the remote nodes. It would also provide a centralized data logging facility and serve as an access point to the network. Because of the relative simplicity of the tasks it must perform, the central computer could itself be a small computer. A back-up computer could then be economically provided. This would reduce the vulnerability of the system to a fault at the top of the hierarchy.

There are many advantages to this approach. Reliability is greatly improved, due to redundancy of processors, hardware, and communication paths. The failure of a processor for example, is not catastrophic to the system, since the rest of the system may continue to function, possibly with slightly degraded performance. Data is potentially more secure than in a centralized data base system, since it is fragmented over a number of independent memories. Access to each data bank is controlled by its processor. This serves to confine errors. Loss or corruption of a data bank is not catastrophic to the system as a whole. This is true even in the case of the central control computer data, since the data is essentially replicated in the remote nodes.

Apart from reliability, the other main advantage is modularity. Since the system consists of a number of identical processor-memory pairs, it can be constructed to meet the present needs of the user, and system performance may be expanded at a later date in relatively small increments, with correspondingly small and smooth cost increments. Also, changes in one node will have minimal effect on other nodes.

5. Advantages of Distributed Systems

Reliability. As stated earlier, reliability results from redundancy of processors, memory and communication paths. Failures then tend to result in "graceful degradation", whereby the remaining elements may take over the functions of the failed module(s). Performance may be degraded, but the system continues to function. Obviously the tasks performed by many of the processors are not transferable since, for example, a section of the plant will usually (but not necessarily) be interfaced to only one processor. In such a case, the malfunctioning section could be manually controlled until a repair has been made.

Another aspect of reliability is that of communication. Because communication paths have intelligent hardware at each end, highly reliable methods can be used to reduce the probability of errors. Also, because there should always be at least two paths between any two nodes, a break in one path would not then terminate communication.

Response. Response time in a distributed system is potentially faster than a multiprogrammed uniprocessor. This is particularly true if a match of one process per processor is made, thus eliminating multiprogramming and the need for context-switching. A processor could then be dedicated to monitoring a time-critical process, to provide an "instant" response. Whilst this fast response time is at the expense of processor efficiency, software is greatly simplified.

Modularity. A high degree of modularity of hardware and software permits large and small systems to be configured with the same basic design, simply by varying the number of modules. Obviously the degree of modularity is highly dependent on the interconnection topology: this is one of the major considerations in selecting a configuration. A modular system can be upgraded in relatively small performance and cost increments. Also, changes can be made in a module with minimal effect on the rest of the system. This is in contrast to a uniprocessor system, where modification or expansion demand extensive system changes, perhaps even replacement of the computer.

Homomorphism. This term, introduced by Jensen, describes the structural correspondence of the multi-process architecture to the multiprocessor architecture. This aspect is responsible for the relative simplicity of the distributed system (at least from this perspective), since software mapping from the virtual to the real architecture is minimal. Reliability is greatly benefitted, since errors are confined to a subset of system functionality and performance [40].

Software. Each processor in a distributed system is controlling one process, or a number of processes of which no more than one is active at any time. Software for such a system will be far simpler than that of a multi-programmed computer. Since system software consists of a number of identifiable modules, it is readily amenable to development by a team. Testing of each module is facilitated. Finally, simplicity in software greatly enhances reliability.

The programmes and data in any one processing-element are relatively isolated from the rest of the system. The probability of illegal interference between processes is thus minimized, if not excluded, because access cannot occur without the knowledge and permission of that processing-element. Data and programmes are thus considerably more secure than in the uniprocessor.

Cost. In a uniprocessor system, the vast network of cables is expensive. A distributed system offers considerable savings in cable costs since all information can be transmitted over a single pair of wires. However, this is complicated by the frequent requirement for manual back-up control from the control room. Also the saving in cable costs is generally a small percentage of total plant cost.

Maintenance. A node can be isolated from the rest of the system in order to perform routine testing and maintenance, without interference to or from the plant.

6. Interconnection Structures

There are obviously many ways in which a number of processors and memory modules can be interconnected. Broadly speaking, multiple-processor systems can be partitioned into tightly- and loosely-coupled systems [18].

Tightly-coupled Systems. A tightly-coupled system is one in which several processors are in close physical proximity. Intercommunication occurs via shared memory or over high-speed parallel buses.

Loosely-coupled Systems. These are systems in which a number of remote computers are connected into a network by data links. There are basically two types of networks [18].

(a) A so-called general purpose network in which each node is capable of independent, stand-alone operation. A typical example is the ARPA network which consists of a number of remote computers. These are linked together into an irregular network to enable sharing of resources [55-58].

(b) A control network of the type discussed earlier. In this system the nodes co-operate in order to achieve overall supervision of a physical system, such as an industrial process.

Interconnection structure is a most important issue in the design of a distributed system, since all other issues tend to be highly dependent on it. In terms of reliability, for example, the question of distribution strategy is critical. In, say, a star-connection, all communication is routed via a single, centralized switch — the whole system is then highly vulnerable to a switch failure.

Due to the current confusion in distributed systems, there is little consensus on the classification of difference topologies. The most widely accepted taxonomy (naming scheme) is that of Anderson and Jensen, shown in Figure 1. The strategy they have adopted in classifying a system is based on the design decisions implicit in the particular configuration [1].

The following section describes the features of various topologies with particular reference to their suitability to real-time plant control.

7. Network Configurations

Fully Connected. Each node is connected to every other node in the network. Cost-modularity is very poor since the addition of one node to an n-node network, requires additional n-connections, one to each node. Complete interconnection is attractive only for very small systems consisting of about three computers. Larger systems are of theoretical interest only.

Partially Connected. For a large number of geographically distributed computers, a partially connected irregular network is most popular. The foremost example is Arpanet which connects over 50 centres spread across North America, with satellite links to London and Hawaii.

Shared Bus. In the shared bus topology, several processing-elements (processor-memory pairs) communicate via a shared bus. Since only one device may transmit over the bus at any one time, bus allocation is of critical importance. Bus control may be centralized by means of some sort of central switch. Decentralized bus control is however preferable from the point of view of reliability, since a failure of the central switch would be catastrophic.

Nodes in a shared bus architecture are an "equal distance" from each other from the point of view of message transfer time. Also, nodes are not distinguished topologically. To increase performance, additional processing-elements may be connected to the bus. However, bus bandwidth is a limiting factor. Unless a fully redundant bus is provided, the system is highly vulnerable to a bus-failure. On the other hand, failure of a processing-element should have minimal effect on system operation, and reconfiguration would simply consist of reallocation of the processes that were being executed by the failed processing-element.

The common bus structure is particularly suited to systems in which executive control is fully decentralized. Honeywell's Modular Computer System is such an example [38, 39]. In the Modular Computer System all inter-process communication is in the form of explicit messages which are transmitted onto the bus, even if the source and destination processes are resident in the same processing-element. This externalization of all inter-process communication not only greatly simplifies software, but also permits all processing-elements to "listen-in" to all conversations. This means that from the control aspect all processing-elements will have the same "view" of the system. This is particularly significant for detection and diagnosis of errors. It also increases the probability that nodes will act in co-operation rather than in conflict. This latter possibility is a real danger in any system in which control is decentralized.

Loops. This topology consists of a number of nodes connected into a loop. Due to the complexity of bi-directional loops over uni-directional loops, traffic in most cases is uni-directional. Messages are placed onto the ring by the source node and circulate around the ring to the destination nodes, being buffered by intermediate nodes. In the Distributed Computing System at the University of California [41-45], a message continues around the loop until it reaches the node which sourced it, where it is removed. All messages thus pass through all nodes. A feature of the Distributed Computing System is that messages are addressed to processes, not to processors. Each "Ring Interface", which is a hardware unit front-ending each computer, has an associative store which holds the names of all the processes currently active in that node. As the message "passes through" the Ring Interface, it checks the message destination name against the process names in its associative store. If there is a match, it copies the message. This technique allows communication to be independent of the number of nodes in the system and allows processes to move freely between processors — this migration being transparent to other processes. The Distributed Computing System is another of the few attempts to decentralise executive control.

Star. Each processing-element is connected by a bi-directional data link to a central switch. The switch accepts a message, performs address translation, and routes the message to its destination. The system is vulnerable to a switch failure; the switch is also a potential bottleneck.

Shared Memory. A system in which two or more processors share access to common memory is termed a "multiprocessor" [4]. Because a single memory constitutes a potential bandwidth problem, it is often fragmented into a number of independent memory modules, to permit several processor-memory accesses to take place simultaneously. There are basically three methods of interconnecting several processors to several memory modules, namely, via a cross-bar switch, or over one or more time shared buses, or by using multi-port memory modules. [3].

The main problem of shared memory structures, is the fact that a processor can access common memory without the knowledge of the other processors. It is possible for a processor to corrupt programmes and data and thereby interfere with the rest of the system. Error confinement is thus a major problem. Complicated protection structures have been devised in an attempt to take care of the problem [30, 35, 36, 49, 52].

8. Communication Concepts

In this section, some basic communication concepts are discussed.

A **message** is a logical unit of information such as a file, a program or an hourly report, which is transferred from one process to another. A message is of variable length, usually with a maximum length which is fairly long (approximately 8k words in Arpanet).

A **packet** is the basic unit of information in the communication subnetwork. Packets may have fixed length, or variable length with a maximum length which is relatively short (approximately 1 000 bits in Arpanet).

A **circuit-switched** network is one in which a complete connection is made between source and ultimate destination before transmission begins. This is the case in a typical telephone network.

A **packet-switched** network is one in which packets are transmitted into the network. The packet contains routing information such as source and destination addresses, as well as error-detection information.

Store-and-Forward. In most packet-switched networks, direct links between all possible senders and receivers do not exist. Packets must therefore pass through intermediate nodes, where they are stored and then forwarded to the next node in the general direction of the destination.

Broadcast Message. A concept that is of considerable value in certain network structures is that of the broadcast message whereby a message is broadcast to all nodes. It is then the responsibility of each node to determine whether the message applies to them. This has the advantage that no routing is required.

Protocol. Protocol is the procedure for the exchange of information between processes. There are four levels of protocol: the process level at which processes communicate; the message level; the packet level; and the hardware level [18]. Here, we are not concerned with the content of messages, only with providing a certain level of confidence that messages will be reliably transported from sender to receiver, with no errors introduced.

Levels of Protocol

(a) **Process Level.** A number of application-orientated processes reside at each node. To enable overall plant control, information such as setpoints, hourly report data, etc., is passed between processes. The information is passed in the form of messages. Messages may also consist of programmes and files which may be transferred to other processors during reconfiguration or to achieve load balancing.

(b) **Message Level.** The message level interfaces between the process level and the communication sub-network. It is the responsibility of the message level to break down long messages into packets. Each packet is then independently routed to the destination node. The message level is then responsible for re-formatting the incoming packets into the original message, which is then passed up to the process level. An error-check field is appended to the message by the sender. If the receiver detects an error in the message, it ignores it, and typically will send off a request to the sender to retransmit the message. If no errors are detected, an "acknowledge message" is sent to the source. This handshake technique gives positive confirmation to the source that the message has reached its destination. The source will retransmit the message if it does not receive a response within a time-out period, with a maximum of about three transmissions. If possible, the retransmissions should take a different route to optimise the chances of the message reaching its destination. The message protocol must be able to cope with error conditions. This involves detection, diagnosis, and recovery. For example, an acknowledge message could get lost, resulting in two identical messages being received by a process. An error in a destination address could send the message to a wrong process. Generally the message header is protected by a separate error-check field, to reduce the probability of undetected errors in the important routing information. If an error is detected in the header then the message is ignored.

(c) **Packet Level.** Due to the difficulty of dealing with long variable length messages (for example, in a forward-and-store network it would be difficult to estimate buffer requirements, and to estimate delays across the network), messages are usually broken up into a number of packets which the network can handle more easily. In a store-and-forward network, each packet is contains its source and destination node. A packet is routed to a node which examines its destination address. If it is not for that node, it is routed to the next node in the general direction of the destination node. This implies a fair degree of intelligence and storage in each node. It also implies that many packets may be circulating at one time. Also, if a message is broken up into a number of packets, these will have to be numbered so that they can be reassembled at their destination into the original message. Each packet might take a different route, so they might arrive out of sequence. As in the case of messages, packets are acknowledged or retransmitted either on request or after a time-out period.

Problems experienced at the message level are also reflected at the packet level. An additional problem can arise — that of deadlock. As there is a finite amount of buffer space at each node, a situation could arise that all buffer space is filled, so that no node can accept another packet. Thus each may wait indefinitely for the others to empty a buffer.

(d) **The Hardware Transmission Level.** This level deals with connection, transmission and signalling methods.

Discussion. Each level interfaces with the level above and below. Provided these interface specifications are adhered to, changes may be made at any level without effect on other levels. Each level must obviously reflect the overall system philosophy.

In certain applications, messages will not be very long and will not need to be broken into shorter packets. The distinction between message and packet levels then falls away. The software/hardware ratio as described here is not necessarily true for all systems. For example, in the Distributed Computing System, the routing function of a node (which consists of copying messages which are destined for local processes, as the message passes through the node), is implemented in hardware. So is the error-checking and acknowledgement of messages. In the Modular Computer System, the hardware is responsible for even more; in fact everything from the message level down. As hardware becomes more powerful and flexible, it is very likely that it will take over much of what has traditionally been done in software.

9. Design Issues

Introduction

It is impossible to consider any single design issue in isolation, since all issues impact on each other. Ultimately, the single criterion is cost — the minimum cost to achieve certain specifications. Because of the present shifting balance of hardware/software costs, it is becoming necessary to reconsider previous decisions. Also, new technology has made possible designs which in the past were not cost-effective. The falling cost of processors in particular has led to fundamental changes in the approach to system architecture in certain application areas. It is no longer necessary to optimise processor usage; a requirement which, in the past, led to large, complicated operating systems.

The most important design decisions are: how to partition the computational load across many processors; how the resulting processes are to communicate; and what interconnection structure is required to support this communication [5].

Load Partitioning

The decomposition of a system into individual, autonomous processes is critical to a successful design. The theoretical approach is of little value at this stage, especially if optimization is attempted [19]. It would appear that the best approach is one based on experience and intuition. Fortunately, most dedicated real-time systems can fairly easily be subdivided into a number of relatively loosely-coupled, well defined sections. Broadly speaking, the objectives of partitioning should be [14].

(a) To segment the system into separate, distinct, autonomous modules;
(b) To minimise interference between modules;
(c) To minimise communication between modules;
(d) To maximise parallelism and concurrency;
(e) To minimise multiprogramming.

Resource Allocation

In terms of hardware, it is obviously preferable that it be composed of identical modules, particularly in the case of processing-elements. Apart from the obvious cost advantages, software would be fully transferable. This considerably facilitates resource allocation, especially if it is to be done dynamically to achieve load balancing or reconfiguration after a fault. Unfortunately, the partitioning procedure is unlikely to yield similarly sized software modules 'naturally'. The larger modules could be

further decomposed, but this procedure will necessarily increase inter-module communication and further increase the problems of synchronizing modules.

Alternatively, the processing-element can be selected so that it could execute the largest software module. This would result in greatly simplified software, at the expense of hardware efficiency. The most cost-effective solution would probably lie somewhere in between. In an hierarchical structure, the requirement for software transportability would be reduced, since higher level software could not normally be executed by lower level hardware, and vice versa.

Process Intercommunication

Processes must communicate in order to achieve overall control of the plant. The first question to ask is: "What do processors in a real-time control plant say to each other?" Hewlett-Packard propose the following answers [7]:

— Deposit and extract data into and out of a distributed or centralized file system;
— Exchange messages directly between executive-level modules in different nodes;
— Share centralized peripherals for reduced overall system cost;
— Share storage of common executable modules for down line loading and execution;
— Dyamically share computing resources as work loads vary.

In a centralized uniprocessor there are many 'protocols', all of them different. Synchronization is achieved by semaphores or flags. Interrupts provide fast response to time-critical service requests. Data is transferred via (common) memory. Device I/0 employs its own protocol. This proliferation of techniques complicates things enormously. In a distributed environment, it could easily be even more complex, due to the relative isolation of processing-elements and the resulting time delays in communication.

In a loosely-coupled system there is a strong argument for adopting a uniform approach to **all** interprocess communication. This is achieved by means of a message-orientated communication system. Input/Output, synchronization, data transfers, etc., all occur by means of explicit messages between the respective processes. Interrupts could be handled directly by hardware where possible or scheduled by hardware together with other messages.

System Control

The resources of any system must be controlled. So far, very little has been said concerning system-wide executive control and yet this is probably the most important issue in distributed systems, especially in the process-control area. Here we discuss this subject only briefly, due to the fact that very little of a practical nature is known regarding fully decentralized control.

There are four elements that can be distributed: hardware (including processors), processing, data and system control [9]. The hierarchical control system, as described above, distributes hardware, processing and data but control of the system is centralized in a single processor. There is no reason, theoretically speaking, why control too, should not be fully decentralized. There are also very good reasons why we should attempt to decentralize control.

In any system in which processors and data are dispersed, considerable problems arise due to the inevitable time lags in transmitting data from one point to another. The system state as seen by the central control processor (in an hierarchical network) will thus always be out of date and therefore inaccurate or even in error. However, in order to make decisions, it must assume that its state information is valid. Thus it may happen that invalid directives are transmitted to processors lower in the hierarchy. These processors which, on the basis of their more up-to-date information, could determine the directive to be faulty, even dangerous, are powerless to refuse it. However, the main weakness of an hierarchical structure is the fact that the whole system is vulnerable to a fault or failure of the central controlling processor or its software. There is, thus an upper bound to the reliability of systems in which there is any degree of centralization of control.

The advantages of fully decentralized control are summarized by Jensen [40].

> "We hypothesize that there is a positive correlation between the amount of decentralized system-wide executive control, and the extent to which the system can achieve certain attributes. Foremost amongst these attributes are — 'extensibility',* 'integrity', and 'performance'."

The concepts of fully decentralized computers have certain fundamental differences to partially decentralized systems. These concepts are briefly discussed in the following section.

10. Fully Distributed Computers

Jensen defines a fully distributed computer to be a "multiplicity of processors that are physically and logically interconnected to form a single system, in which overall executive control is exercised through the co-operation of decentralized system elements" [40].

There are essentially five components to this definition [9]:

(a) The system consists of a multiplicity of physical and logical components that can undertake specific tasks on a dynamic basis to achieve load balancing, or to permit reconfiguration of the system after failure of a subset of its elements.
(b) These resources are physically distributed. Each processor has its own operating system and acts as an autonomous entity. The system components interact via a communication network, which employs a two-party co-operative protocol (as opposed to a two-party master-slave protocol) to achieve transfer of information.
(c) The distributed components are integrated into a single, logical, cohesive entity by means of a high-level operating system, which manages all of the system's physical and logical resources. Each processor may have its own operating system which may be unique. Alternatively, as in the Modular Computer System, each processor may have a copy of kernel logic. These copies execute in parallel in a non-hierarchical fashion. Taken as a whole, these kernel copies constitute the high-level operating system.
(d) System structure is transparent to the user. Services are requested from the high-level operating system by name only and the server does not have to be identified. The user may use any of the physical or logical resources of the system, as if these resources were locally available.
(e) The interaction of all physical and logical resources is that of co-operating, autonomous elements. Master-slave relationships are excluded. This is for the obvious reason that a slave is powerless to refuse directives which it could determine to be faulty or undesirable by virtue of its better knowledge of local conditions. It is thus important that the destination resource should be able to refuse a message or reject a request for service based on the knowledge of its own status.

Fully distributed computers are beyond the current state-of-the-art. Nevertheless, there is intense interest in the possibilities of such systems.

Operating Systems

In any multiple-processor configuration, up-to-date status information is never available at one point. This is due to the inevitable time delays that occur when data is transmitted from one point to another. The operating system must therefore be designed to work with inaccurate or even erroneous status information. This is in contrast to a centralized

*'extensibility' is approximately equivalent to 'modularity'

system where the operating system is assumed to have access to complete and accurate information about the overall system status.

The high-level operating system should exercise control over all of the system's resources. This task is greatly simplified if the system is hierarchical. However, the requirement for strictly non-hierarchical control (that is, there are no master-slave relationships) greatly exacerbates the control problems. Even if the multiple autonomous processes are designed to co-operate, the likelihood of conflicting action is much higher than in an hierarchy.

Synchronization of the system as a whole is complicated by the time lags which are inevitable when two autonomous processes attempt to communicate. The conventional methods used in uniprocessors to synchronize two processes such as flags, semaphores, wake-ups, etc., cannot directly be used, although for example, a message carrying the semaphore can be passed between the processes. This consumes a great deal of processing time quite apart from communication delays. A hardware mechanism of some sort might feasibly reduce delay.

Management of resources is a complex problem. For the system to function successfully, efficient control must be exercised over each and every element of the system. Each component of the system cannot be viewed as an isolated entity, but its relationship with the rest of the system must be carefully considered. Dynamic allocation of resources is thus a complex task, particularly if this is being done as a result of failure of some elements of system. This then impacts on the underlying control algorithm for the plant since changes in the overall control strategy would be required.

11. Conclusion

The field of distributed processing is one of considerable activity at the present time. However, we should not expect immediate, dramatic changes. It will be many years before the full potential of distributed systems is exploited.

12. Acknowledgements

The author wishes to acknowledge the many writers who are responsible for the ideas expressed in the text, Douglas Jensen in particular. He also expresses his sincere appreciation to Ken Behr for his assistance in production of the diagram, and to Mike Rodd for editing the article and for many of the ideas expressed therein. The support of the National Institute for Metallurgy and the University of Cape Town is gratefully acknowledged.

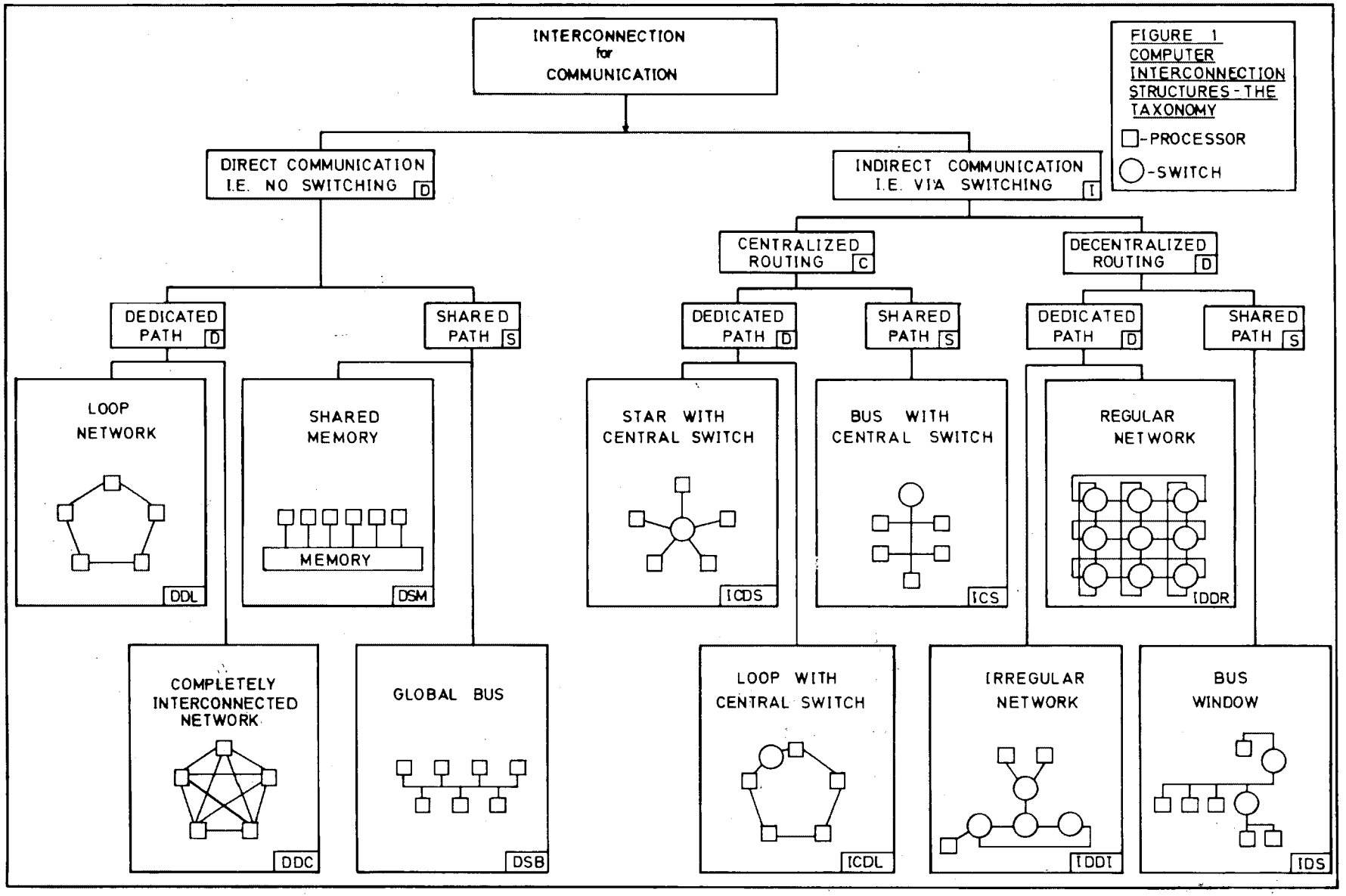

INTERCONNECTION for COMMUNICATION
FIGURE 1 COMPUTER INTERCONNECTION STRUCTURES - THE TAXONOMY
□-PROCESSOR
○-SWITCH
DIRECT COMMUNICATION I.E. NO SWITCHING D
INDIRECT COMMUNICATION I.E. VIA SWITCHING I
CENTRALIZED ROUTING C
DECENTRALIZED ROUTING D
DEDICATED PATH D
SHARED PATH S
DEDICATED PATH D
SHARED PATH S
DEDICATED PATH D
SHARED PATH S
LOOP NETWORK
DDL
SHARED MEMORY
MEMORY
DSM
STAR WITH CENTRAL SWITCH
ICDS
BUS WITH CENTRAL SWITCH
ICS
REGULAR NETWORK
IDDR
COMPLETELY INTERCONNECTED NETWORK
DDC
GLOBAL BUS
DSB
LOOP WITH CENTRAL SWITCH
ICDL
IRREGULAR NETWORK
IDDI
BUS WINDOW
IDS

Figure 1

References

Reviews

1. ANDERSON, G.A. and JENSEN, E.D. (1975). Computer Interconnection Structures : Taxonomy, Characteristics, and Examples. *ACM Computing Surveys,* **7,** 197-213.
2. CASAGLIA, G.F. (1976). Distributed Computing Systems : A Biased Review, in *Euromicro,* North-Holland, Amsterdam, 5-18.
3. ENSLOW, P.H. (1977). Multiprocessor Organization — A Survey, *ACM Computing Surveys,* **9,** 103-129.
4. HOOGENDOORN, C.H. (1977). Multiprocessor Systems : Potentials and Problems, paper presented at Program 77 Conference.
5. JENSEN, E.D. (1975). The Influence of Microprocessors on Computer Architecture — Distributed Processing, in *Proc. ACM. Nat. Conf.,* 125-128.
6. RODD, M.G. (1977). A Survey of Distributed Computer Systems with special reference to Industrial Control Applications, Internal Report, University of Cape Town.

Design Issues

7. DICKEY, S. (1976). Distributed Computer Systems — Interfaces, Internals and Inferences in *Infotech State-of-the-Art Report on Distributed Processing,* 225-242.
8. DIJKSTRA, E.W. (1974). Self-Stabilizing Systems in spite of Distributed Control, *Comm. ACM,* **17,** 643-644.
9. ENSLOW, P.H. (1976). What does 'Distributed Processing' Mean?, in *Infotech State-of-the-Art Report on Distributed Processing,* 259-272.
10. LAMPSON, B.W. (1973). A Note on the Confinement Problem, *Comm ACM,* **16,** 613-615.
11. LE LANN, G. (1977). Distributed Systems — Towards a Formal Approach, in *Information Processing 77,* North Holland, Amsterdam, 155-160.
12. METCALF, R.M. (1972). Strategies for Interprocess Communication in a Distributed Computing System, *Proc. Symp. Computer-Communications, Networks and Tele-Traffic,* Polytechnic Institute of Brooklyn.
13. PARNAS, D.L. (1972). On the Criteria to be used in Decomposing Systems into Modules, *Comm. ACM,* **15,** 1053-1058.
14. RAMAMOORTHY, C.V. and KRISHNAROO, T. (1976). The Design Issues in Distributed Computer Systems, in *Infotech State-of-the-Art Report on Distributed Processing,* 377-399.
15. ROBERTS, L.G. (1977). Packet Network Design — The Third Generation in *Information Processing 77,* North-Holland, Amsterdam.
16. SIEWIOREK, D.P. Modularity and Multi-Microprocessor Structures, Dept. of Computer Science and Electrical Engineering, Carnegie-Mellon University, Pittsburgh.
17. SIEWIOREK, D.P. and BARBACCI, M.R. (1976). Modularity and Multi-processor Structures — Some Open Problems in the Construction and Utilization of Mini- and Micro-Processor Networks, in *Infotech State-of-the-Art Report on Distributed Systems,* 403-418.
18. SLOMAN, M.S. and PENNEY, B.K. (1977). Communication Requirements for Microcomputer Networks in Process Control Applications, internal report, Dept. of Computing and Control, Imperial College, London.
19. WHITE, G.N.T. and SIMMONS, M.D. (1977). Analysis of Complex Systems, University of Cambridge, England.

Software

20. ABRAMSON, N. and KUO, F.F. (1973). *Computer Communication Networks,* Prentice-Hall, Englewood Cliffs.
21. ANDERSON, G.A. Interconnecting a Distributed Processor System for Avionics, in *Proc. 1st Annual Symp. on Computer Architecture,* 11-16.
22. BRINCH-HANSEN, P. (1970). The Nucleus of a Multiprogramming System, *Comm ACM,* **13,** 238-250.
23. DAGLESS, E.L. A Multimicroprocessor: CYBA-M, Dept. of Electrical and Electronic Engineering, Univ. College of Swansea.
24. DAVIES, D.W. and BARBER, D.L.A. (1973). Communication Networks for Computers, John Wiley and Sons.
25. DIJKSTRA, E.W. (1968). The Structure of 'THE' Multiprogramming System, *Comm ACM,* **11,** 341-346.
26. DIJKSTRA, E.W. (1970). Notes on Structured Programming, Report 70 — WSK — 03, Technical University, Eindhoven.
27. FARBER, D.J. (1974). Software Considerations in Distributed Architectures, *Computer,* March 1974, 31-35.
28. HARMALA, A.D. (1975). Benefits of Localized Control with microcomputers, *Computer Design,* May 1975.
29. HEWLETT-PACKARD JOURNAL (1978). Articles on the HP Distributed System Network, March 1978.
30. LAMPSON, B.W. (1969). Dynamic Protection Structures, in *Proc. AFIPS FJCC,* 27-38.
31. McFADYEN, J.H. (1976). Systems Network Architecture : an Overview, *IBM Systems Journal,* **15,** 1, 4 - 22.
32. METCALF, R.M. and BOGGS, D.R. (1976). Ethernet: Distributed Packet Switching for Local Computer Networks, *Comm ACM,* **19,** 395-403.
33. MOORE, M.J. (1974). A Distributed Microprocessor System for Avionics, 8th Asilomar Conference on Circuits, Systems and Computers, Dec. 3-5 1974, *Calif. IEEE,* 92-96.
34. MORGAN, D.E. and TAYLOR, D.J. (1977). A Survey of Methods of Achieving Reliable Software, *Computer,* Feb. 1977, 44-53.
35. POPEK, G.J. (1974). Protection Structures, *Computer,* June 1974, 22-23.
36. WULF, W.A. (1975). Reliable Hardware/Software Architecture, *IEEE Trans. on Software Engineering,* **SE-1,** No. 2.
37. YAN, G. and L'ARCHEVEQUE, J.V.R. (1976). On Distributed Control and Instrumentation Systems for Future Nuclear Power Plants, *IEEE Trans. Nuclear Science,* **NS-23,** 431-435.

The Modular Computer System

38. JENSEN, E.D., A Distributed Function-Computer for Real-time Control, in *Proc. 2nd Annual Conf. on Computer Architecture,* ACM, 176-182.
39. JENSEN, E.D. (1976). Distributed Processing in a Real-time Environment, in *Infotech State-of-the Art Report on Distributed Processing,* 305-318.
40. JENSEN, E.D. (1978). The Honeywell Experimental Distributed Processor — An Overview, *Computer* Jan. 1978, 23-38.

The Distributed Computing System

41. FARBER, D.J. (1975). A Ring Network, Datamation, Feb. 1975, 44-46.
42. FARBER, D.J. and HEINRICH, F.R. (1972). The Structure of a Distributed Computing System — The Distributed File System, in *1st International Conf. of Computer Communications,* ACM/IEEE, Oct. 24-26.
43. FARBER, D.J. and LARSON, K.C. (1972). The System Architecture of the Distributed Computer System — The Communications System, in *Proc. Symp on Computer-Communications Networks and Teletraffic,* Polytechnic Institute of Brooklyn, 21-27.
44. FARBER, D.J. and LARSON, K.C. (1972). The Structure of a Distributed Computing System — Software, in *Proc. Symp. on Computer-Communications Networks and Teletraffic,* Polytechnic Institute of Brooklyn, 539-545.
45. ROWE, L.A. *et. al.* Software methods for Achieving Fail-Soft Behaviour in the Distributed Computing System, Dept. of Information and Computer Science, Univ. of California, Irvine.

C.mmp

46. BELL, C.G. and FREEMAN, P. (1972). C.ai — A Computer Architecture for AI Research, in *Proc. AFIPS FJCC,* 779-790.
47. FULLER, S.H. (1976). Price Performance Comparison of C.mmp and the PDP-10, in *3rd Annual Symp. on Computer Architecture,* Jan. 19-21, 1976, ACM.
48. WULF, W.A. and BELL, C.G. (1972). C.mmp — A Multi-Mini-Processor, in *Proc. AFIPS FJCC,* 765-777.
49. WULF, W., COHEN, E., CORWIN, W., JONES, A., LEVIN, R., PIERSON, C. and POLLACK, F. (1974), HYDRA — The Kernel of a Multiprocessor Operating System, *Comm ACM,* **17,** 337-345.
50. WULF, W. and LEVIN, R. (1975). A Local Network, *Datamation,* 47-50.

CM*

51. FULLER, S.H. *et al.* Computer Modules — An Architecture for Large Digital Modules, in *Proc. 1st Annual Symp. on Computer Architecture,* ACM, 231-237.
52. JONES, A.K. *et al.* (1977). Software Management of CM* — A Distributed Multiprocessor, in *Proc. AFIPS NCC,* 657-663.
53. SWAN, R.J. *et al.* (1977). CM* — A Modular, Multimicroprocessor, in *Proc. AFIPS NCC, 637-644.*
54. SWAN, R. J. *et al.* (1977). The IMplementation of the CM* Multimicroprocessor, in *Proc. AFIPS NCC,* 645-655.

Arpanet

55. CARR, C.S. *et al* (1970). Host-Host Communication Protocol in the ARPA network, in *Proc. AFIPS SJCC,* 589-597.
56. HEART, F.E. *et al* (1970). The Interface Message Processor for the ARPA Computer Network, in *Proc. AFIPS SJCC,* 551-567.
57. ORNSTEIN, S.M. *et al* (1975). Pluribus — A Reliable Multiprocessor, in *Proc NCC,* 551-559.
58. THOMAS, R.H. (1973). A Resource Sharing Executive for the Arpanet, in *Proc. NCC,* 155-163.

The P-NP Question and Recent Independence Results

N.C.K. Phillips

Dept. of Mathematics, University of the Witwatersrand, Johannesburg, South Africa.

Abstract

Despite intensive research the P = NP question is unresolved and the research suggests that it is hard to answer. The corresponding question for query machines with recursive oracles is undecidable in set theory. Simply determining whether a procedure halts or the running time of an algorithm may be harder than we expect. There is a Turing machine which does not halt yet its halting is undecidable in set theory. There is an algorithm which runs in time n^2 yet it cannot be proved in set theory to run in any time less than 2^n.

1. The classes P and NP

It is commonly accepted that an algorithm is manageable if its running time is bounded by a polynomial function of the length of its input. However there are many computational problems for which the running times of the known algorithms are bounded only by exponential functions. One such problem is that of determining whether a formula of the propositional calculus in CNF (conjunctive normal form) is satisfiable. This is known as the Satisfiability problem. One algorithm for Satisfiability is to compute the truth table for the formula — but if the formula contains n literals this requires 2^n steps. The travelling salesperson problem (known as the travelling salesman problem before male chauvinism became unpopular) and determining the chromatic number of a graph are two more of the many problems whose known algorithms are not polynomial bounded. Clearly it is of practical use to ask whether these problems do have algorithms whose running time is polynomial bounded.

Through simple encodings into sets of words over a finite alphabet many computational problems can be converted into language recognition problems. Details of such encodings can be found in [1]. We shall assume that we are encoding into the set Σ^* of strings over the alphabet $\Sigma = \{0,1\}$. In particular natural numbers are to be encoded by their binary representations. Sometimes there are alternative "natural" encodings. For example we can represent a graph by an encoding of its adjacency matrix or by encoding its arcs as pairs of adjacent nodes. For our purposes the detail of such encodings will not be important since it normally turns out that if one natural encoding of a problem yields a language (i.e. a subset of Σ^*) recognizable in polynomial time then so will another natural encoding.

We should be precise about what we mean by a language recognition algorithm. Any of the standard mathematical formulations will do but we shall usually think of such an algorithm as a Turing machine. For the purpose of deciding whether an algorithm is polynomial bounded it does not matter whether we think of single tape or multitape Turing machines, because a language recognized in polynomial time p(n) by a multitape machine is recognized in time $p^2(n)$ by a single tape machine — see [1].

We shall need the notion of a nondeterministic algorithm. This may be thought of as a procedure which at some steps has a finite number of choices for its next step, or as a modified Turing machine which in some states has alternative choices for its next state. A precise definition can be found in [1]. On a given input a nondeterministic machine will perform one of several possible computations. An input is said to be accepted by such a machine if among its possible computations there is at least one which leads to an accepting state.

Now for many computational problems we do not know polynomial bounded algorithms but we do have polynomial bounded nondeterministic algorithms. Satisfiability is such a problem. We can guess truth values for the literals in a formula then check in polynomial time whether these guesses make the formula true. Whether a graph can be k-coloured is another such problem. We can guess an assignment of colours to the nodes and then check in polynomial time whether no two adjacent nodes have the same colour. Our discussion so far motivates the following definitions.

Definitions 1. P is the class of languages over Σ recognizable by Turing machines in polynomial bounded time.

2. NP is the class of languages over Σ recognizable by nondeterministic Turing machines in polynomial bounded time.

When we say that a problem is in P (or NP) we mean that a natural encoding of it is in P (or NP).

2. The P - NP question

Definition (Karp [7]) Let L,M be languages over Σ. L is said to be reducible to M iff there is a function $f: \Sigma^* \to \Sigma^*$ which is computable in polynomial time such that, for all x, $x \in L$ iff $f(x) \in M$.

The important point about this definition is that if L is reducible to M and $M \in P$ then $L \in P$.

Theorem (Cook [3]) If $L \in NP$ then L reduces to (the encoding of) the Satisfiability problem.

In view of the wide extent of the class NP, Cook's theorem is remarkable. It implies that if the Satisfiability problem has a polynomial bounded algorithm then so does every problem in NP. We state this as a corollary.

Corollary P = NP iff Satisfiability $\in$ P.

Definition L is said to be NP complete iff 1. $L \in NP$

2. P = NP iff $L \in P$.

In the language of this definition, Cook's theorem implies that Satisfiability is NP complete. It is again remarkable that many NP problems are NP complete. To show that an NP problem is NP com-

Received October 1978

plete one need only show that Satisfiability reduces to it. The first extensive list of such problems appeared in Karp [7]. We should mention that there are NP problems which are not known to be NP complete.

It makes sense to investigate whether P = NP, particularly if there is evidence that these classes are not far apart. The early evidence was quite startling. Let n-Satisfiability denote the problem of determining whether a formula in CNF which has at most n literals per clause is satisfiable.

Theorem (Cook [3]) 2-Satisfiability $\in$ P. 3-Satisfiability is NP complete.

One approach to the P = NP question has been to seek "simpler" NP complete problems — for example see [10]. Another approach has been to seek good polynomial bounded algorithms for optimization versions of NP complete problems. Some approximate algorithms perform extremely well. For the Subset-Sum problem there is a sequence of polynomial bounded approximate algorithms which asymptotically approach the optimal solution — see [6]. In contrast, unless P= NP there is no polynomial bounded algorithm which gives a good approximation for the NP complete problem of finding the chromatic number of a graph — see [4].

Slightly varying a problem can have a drastic effect on our knowledge of how well it can be approximated by polynomial bounded algorithms. A good example of this is the travelling salesperson problem (which is NP complete).

Theorem [8] There is a polynomial bounded algorithm which, for any instance of the travelling salesperson problem, will produce a tour of length less than twice the optimal length.

However, let us change the travelling salesperson problem slightly by no longer requiring that the distance function d(i, j) (which gives the distance between towns i and j) should satisfy the triangle inequality d(i, j) $\leq$ d (i, k) + d (k, j).

Theorem [9] Let N be any positive integer. Then P = NP iff there is a polynomial bounded algorithm which, given any instance of the modified travelling salesperson problem, will produce a tour of length less than N times the optimal length.

A further approach to the P = NP question has been to change the classes P and NP slightly and see what happens. To this end we need the idea of a query machine. A query machine is a multitape Turing machine with a distinguished worktape called a query tape and three distinguished states a query state, a yes state and a no state. A query machine has a set, called an oracle, associated with it. On entering the query state the next move of a querry machine with oracle X is to enter the yes state if the string on the query tape belongs to X and to enter the no state otherwise. A query machine is said to be polynomial bounded if there is a polynomial p(n) such that on every input of length n and with every oracle the query machine halts within p(n) steps. P^X is the class of languages over Σ recognized by polynomial bounded query machines with oracle X. NP^X is the class of languages over Σ recognized by polynomial bounded nondeterministic query machines with oracle X.

Theorem [2] There are recursive sets A and B such that $P^A = NP^A$ and $P^B \neq NP^B$.

This theorem shows that we will not be able to prove that P = NP or that P $\neq$ NP by any technique which would carry over to query machines.

3. Independence Results

The P = NP question has proved to be hard to answer. It may even be undecidable. We shall show that the corresponding question for query machines with recursive oracles is undecidable in Zormelo-Fraenkel set theory ZF (we shall of course assume that ZF is consistent). Perhaps even the problem of determining the running time of an algorithm is harder than we expect. Many of us feel that, given an algorithm and sufficient ingenuity, we should be able to determine its running time. If we feel this way we shall be surprised by the result that there is an algorithm which runs in time n^2 but which cannot be proved in ZF to run in time less than 2^n. These results are due to Hartmanis and Hopcroft [5].

Lemma If B Δ C (the symmetric difference of B and C) is finite then $P^B = P^C$ and $NP^B = NP^C$.

Proof Suppose that D is a finite set and that we have a query machine which is bounded by a polynomial p(n). If we modify this machine so that each time it queries it must afterwards check whether the string on the query tape belongs to D then this new machine is bounded by p(n) + O (p(n)) = O (p(n)). If B Δ C is finite then two such modifications will enable us to simulate a polynomial bounded query machine with oracle B by a polynomial bounded query machine with oracle C and vice-versa. So $P^B = P^C$. Similarly $NP^B = NP^C$.

Let $\{\phi_i\}$ be an effective enumeration of single tape Turing machines. We can apply Kleene's s-m-n and recursion theorems to such an enumeration (these theorems can be found in [11]). By P^{ϕ_i} we shall mean P^C where C is the set recognized by ϕ_i and NP^{ϕ_i} will denote NP^C.

Theorem There is a recursive set C such that $P^C = NP^C$ is undecidable in ZF.

Proof Let A,B be recursive sets such that $P^A = NP^A$ and P^B NP^B. Let ϕ be a Turing machine which accepts an input (x,j) and halts if either there is a proof among the first x proofs of ZF that

$$P^{\phi_j} = NP^{\phi_j} \text{ and } x \in B,$$

or there is a proof among the first x proofs of ZF that

$$P^{\phi_j} \neq NP^{\phi_j} \text{ and } x \in A,$$

and which otherwise rejects the input and halts.

By the s-m-n theorem there is a recursive s such that

$$\phi_{s(j)}(x) = \phi(x,j) \text{ for all } x,j.$$

By the recursion theorem there is an n such that

$$\phi_{s(n)}(x) = \phi_n(x) \text{ for all } x.$$

Hence $$\phi_n(x) = \phi(x,n) \text{ for all } x.$$

Let C be the set recognized by ϕ_n. Assume there is a proof in ZF of $P^C = NP^C$ — this is say the m'th proof in ZF.

If $x > m$ then ϕ_n accepts x iff ϕ accepts (x,n) iff $x \in B$. Thus $B \Delta C$ is finite so by the lemma $P^C \neq NP^C$ is a theorem of ZF because $P^B \neq NP^B$ is a theorem of zf. This contradicts our assumption. Similarly the assumption that $P^C \neq NP^C$ is a theorem of ZF leads to the contradiction that $P^C = NP^C$ is a theorem of ZF.

So $P^C = NP^C$ is undecidable in ZF.

Intuitively we know that the set C recognized by ϕ_n in the theorem is the empty set, so that $P^C = NP^C$ iff P = NP. This suggests (but does not prove) that P=NP may be undecidable in ZF.

Note that the properties of ZF needed in the above theorem are that it is consistent, powerful enough to prove all the results needed about algorithms and that its theorems are recursively enumerable. The theorem would apply to any formal system with similar properties. These remarks apply too to the lemma and the theorem which follow. Let $\phi_i(-)$ denote the computation of the ith Turing machine when started with blank input.

Lemma There is an m such that the halting of $\phi_m(-)$ is undecidable in ZF but nevertheless $\phi_m(-)$ does not halt.

Proof Let ϕ be a Turing machine which, on input (x,j), halts if there is a proof in ZF that $\phi_j(-)$ does not halt, and which does not halt otherwise. Again the s-m-n and recursion theorems give an m such that $\phi_m(x) = \phi(x,m)$ for all x. $\phi_m(-)$ halts iff $\phi(-,m)$ halts iff there is a proof in ZF that $\phi_m(-)$ does not halt. Hence $\phi_m(-)$ does not halt but it is undecidable in ZF whether $\phi_m(-)$ halts.

Theorem There is an algorithm which runs in time n^2 but for which there is no proof in ZF that it runs in time less than 2^n.

Proof For each j let ϕ_{j*} denote a Turing machine which, on input n, simulates $\phi_j(-)$ for n steps and then halts in a total of n^2 steps if $\phi_j(-)$ had not halted, 2^n steps if $\phi_j(-)$ has halted. Then : $\phi_j(-)$ does not halt iff $\phi_{j*}(n)$ halts in less than 2^n steps for all n. Let ϕ_m be as in the last lemma. $\phi_m(-)$ does not halt so ϕ_{m*} runs in time n^2. There is no proof in ZF that ϕ_{m*} runs in time less than 2^n since this would prove in ZF that $\phi_m(-)$ halts.

We conclude with a quotation from [5]: "What this suggests is that our inability to settle questions like the P = NP problem or prove lower bounds (for running times of algorithms) may be a consequence of the power (or weakness) of formal systems such as set theory. Clearly an exciting result would be to discover a natural instance of such a problem".

References

1. AHO, A.V., HOPCROFT, J.E. and ULLMAN, J.D. (1974). *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, Mass.
2. BAKER, T., GILL, J. AND SOLOVAY, R. (1975) Relativizations of the P = ?NP Question, *SIAM J. Computing,* **4,** 431-442.
3. COOK, S. (1971). The Complexity of Theorem-Proving Procedures, *Proc. Third Annual ACM Symposium on the Theory of Computing,* Shaker Heights, Ohio.
4. GAREY, M.R. and JOHNSON, D.S. (1976). The Complexity of Near-optimal Graph Coloring, *J. ACM,* **23,** 43-49.
5. HARTMANIS, J. and HOPCROFT, J.E. (1976). Independence Results in Computer Science, Sigact News, December 1976.
6. JOHNSON, D.S. (1974). Approximation algorithms for combinatorial problems, *J. Computer and System Sciences,* **9,** 256-278.
7. KARP, R.M. (1972). Reducibility Among Combinatorial Problems, in *Complexity of Computer Computations* (Ed. R.E. Miller and J.W. Thatcher), Plenum Press, New York.
8. ROSENKRANTZ, D.J., STEARNS, R.E. and LEWIS, P.M. (1974). Approximate Algorithms for the Travelling Salesperson Problem, *Proc. 15th Annual IEEE Symposium on Switching and Automata.*
9. SAHNI, S. and GONZALEZ, T. (1976). P-Complete Approximation Problems, *J. ACM,* **23,** 555 - 565.
10. STOCKMEYER, L. (1973). Planar 3-colorability is Polynomial Complete, Sigact News, July 1973.
11. YASUHARA, A. (1971). *Recursive Function Theory and Logic,* Academic Press.

Text Compression Techniques

J. E. Radue

Department of Computer Science, University of Natal, Durban.

Abstract

The benefits associated with text compression include more efficient use of peripheral devices, faster information transfer rates and, in some cases, improved sorting speeds through the use of shorter sort keys. However, these advantages must be balanced against a slight increase in CPU-time and the extra storage required for the associated code tables.

Information theory and statistics of English provide a background for the discussion of various text compression algorithms. The common objective of the methods described is to reduce the physical size of the text file while maintaining a complete representation of the information (reversible compression). The methods can be divided into two main classes:

(a) those that re-define the symbol codes to more accurate reflect the information content of each symbol, and

(b) those that use special codes to represent commonly occurring groups of symbols, thereby reducing redundancy due to mutual information between symbols.

Compression techniques not covered include those dealing with data files and with telemetry.

Finally, another approach to compression is described, which also holds some promise for automatic indexing and simpler inverted file design in document retrieval systems.

1. Introduction

Although the cost of both immediate access and secondary memory seems to be continually declining, it is still sound sense to use storage economically. Efficient storage, or compression, techniques currently implemented generally result in a reduction, by a factor of about 2, in the storage required for a given amount of data. Put another way, using the same storage space, double the amount of data can be recorded. This is, however, not the hoped for panacea of the "need more disc-space" syndrome, as the compression algorithms do require some CPU time, and some data files may not be amenable to compression.

The common objective of the methods to be described here is to reduce the physical size of the text file while maintaining a complete representation of the information — termed reversible compression. Text compression, by which is meant reversible compression of documents or books written in a natural language, will be dealt with exclusively.

Other implications of compressing data, or text, include:

- **reduced data transmission costs**
  Much work is being done in the area of linking computers into networks, and the compression algorithms used include error detection and correction facilities. (These algorithms will not be dealt with here). However, even reducing the amount of useful information being transferred from secondary to main memory in one computer system could justify the use of compression.
- **sorting**
  Some forms of coding can achieve efficient compression while maintaining lexicographic ordering. The resultant shorter keys (and records) should speed up sorting, while data transmission time is also reduced.
- **distributable tapes**
  Computer-readable databases are currently distributed by abstracting and indexing services to act as the source for SDI services. If these are produced in compressed form, the encoding cost is incurred only once, while the subsequent cost of producing, and distributing, duplicate tapes is reduced through the use of smaller tapes. In the future, these databases will be distributed electronically within a network, where compression will still be advantageous.

2. Background

Any system that deals with natural language should take into account certain language characteristics. The properties of interest here are all obtained from studies in statistical linguistics, and are mainly formulae which have been found to describe experimentally obtained data.

The oldest vocabulary relation was popularised by Zipf [50], and has been the subject of many arguments and refinements (for example Mandelbrot [22] and Kucera and Francis [17]. "Zipf's Law" is as follows:

$$p_r = 0,1/r$$

where p_r is the relative frequency (or probability) of a word of rank r (Note: The types, or different words, in a sample of free text are ranked in order of decreasing frequency).

Despite criticisms, Zipf's Law is useful because of its simple form and because it is found to hold with useful accuracy for a variety of languages when sample sizes of the order of 100 000 tokens are considered. Zipf's Law indicates that most tokens in a text consist of a small core of types, and in fact counts indicate that, for example, the 64 most frequent types in a sample of English text represents about 50% of the total tokens (Lesk [18].)

It has also been shown (Miller et al [25] and in Kucera and Francis [17] that if the relative frequencies of tokens of the same length are plotted against the logarithm of their respective length, a normal distribution is obtained. A relationship between the number of types and tokens in a large corpus has also been found experimentally (see Mandelbrot [23] and Schipma [34], and is as follows:

$$D = K * T^b$$

where D is the number of word types, T is the number of tokens, and K and b are vocabulary dependent constants, 0 b 1.

Information theory was developed by Shannon [42], and was soon applied to various fields of science. Shannon himself related information theory to linguistics (Shannon [43]. For a language in which characters occur independently, Shannon may be followed by defining the entropy, or the average amount of information per character in a large data set, as

$$H = -\sum_i p_i \log_2 p_i$$

This paper was presented at the S.A. Computer Society Program 77 conference in Johannesburg.

where p_i is the relative frequency, or probability, of the i'th character and the summation is over the different characters in the character set.

When material is represented in binary code in the most efficient way, it can be shown (Shannon [43]) that this entropy will be the average number of binary digits required to encode each character. H is a symmetrical convex function with its maximum at the point

$$p_1 = p_2 = \ldots = p_n = 1/n$$

In other words, the entropy function has a maximum value when all characters in the alphabet are equiprobable. This maximum value is $\log_2 N$ bits, for an N-character alphabet. Thus an efficient coding technique would assign 4,75 bits for a twenty-seven character alphabet.

This means that if a set of statistically independent messages is to be stored, the encoding which minimises storage space will be one that maximises the entropy, and that this occurs when the symbols of the encoding alphabet occur with as near as possible to equal probabilities. Huffman [14] has given a procedure for constructing such an optimum encoding, given a fixed set of messages (i.e., character probabilities are fixed). This procedure will be discussed later.

The characters that occur in natural language text are neither independent (Burton and Licklider [4]) nor equiprobable (Pratt [29]). For example, some letters (in English, E, T, O, A, N, etc.), occur much more frequently than the rest of the alphabet, some letters tend to follow others, e.g. U after Q, H after T, etc., and finally entire words or phrases tend to follow other words. Because these constraints extend over many letters (Burton and Licklider [4]), it is difficult to account for all of them. Shannon [43] estimates that the entropy for a large sample of English text (26 letters plus a blank), is about 1,4 bits per letter. That is, instead of the 4,75 bits of information which could be conveyed by 27 equiprobable, independent characters, English text only contains about 1,4 bits of information per character. This difference represents the redundancy of English, some of which can be eliminated by appropriate encoding schemes. Shannon [43] also showed that the entropy of English text varied with message length:

Message length (chars.)	1	2	3	1 word + space
Entropy (bits/char.)	4,11	3,32	3,10	2,18

Further estimates, by other investigators, of the entropy of a word in English text vary from 1,7 to 2,0 bits/character (Schwartz and Kleiboemer [41].

The number of bits required to represent a character in a computer leads to a further difference. The most widely used character size is 8 bits (1 byte), which theoretically means that for each character in an English text corpus, 8 bits are being used to convey only 1,4 bits of information.

3. Discussion of Algorithms

From the discussion above, it can be seen that there are two basic principles involved in compression. Firstly, it may be possible to more accurately match the compression code to the information content of the input characters and secondly, the input characters may be clustered in some form (e.g., words bigrams or other fragments) to take advantage of the dependencies between characters. However, there is a vast "gray area" which consists of combinations of methods using both these general principles, and methods specially tailored to suit the peculiarities of the text or data being compressed. As a result, the following algorithms will be grouped under similarity of total method, rather than under one of the above two principles (Stoneburner [45]).

Although statistical techniques of compressing digital data will not be dealt with here, they are of fundamental importance in the transmission of data (e.g., from satellites). Typically, these techniques include both compaction and error corection and are not reversible. The ad hoc techniques that depend only on the type of data being compressed, or on the file design, will also not be discussed (these latter techniques include datum subtraction or differences, abbreviations, etc.).

It must be kept in mind that there are tradeoffs in the memory space required to store the code tables and in the time required to perform the appropriate transformation, and that some techniques are better suited to static files than to dynamic ones. It should also be noted that comparisons of the compression achieved by the various techniques are very difficult because of the different source files used.

3.1 Fixed Length Coding for Character Strings

3.1.1. Character Repeat Suppression

Although this technique works best on files which are known, in advance, to have long strings of repeating characters (such as program source files with strings of blanks), it is mentioned briefly here as the method is simple, execution is fast, memory requirements are small and appreciable compression can sometimes be achieved.

It consists of replacing a string of repeating characters with a code which describes the string composition. This replacement code is usually three characters long, the first being a unique, or special character (which is relatively rare in the file to be compressed), indicating the start of the code. The next two characters indicate, respectively, the character being suppressed and the repeat count (usually binary) of that character. Obviously only repeats of characters longer than the replacement code should be encoded by this method. If the special character is encountered in the data, it could be encoded as a repeat count of 1.

A slight variation in the technique is used where it is known that many of the repeated strings consist of only a few characters, such as blanks or zeros. A different special character is used for each of these common repeated characters, resulting in a replacement code of only two characters — the indicator and the count.

3.1.2. Bigram Substitution

This technique makes use of the fact that, for some code sets (such as EBCDIC), the number of bit codes available is a great deal larger than the number of characters in the standard character set. The unused codes are substituted for the more frequently occurring bigrams. Theoretically, the maximum reduction achievable is 50%, since, at best, one character substitutes for a bigram (Bookstein and Fouty [3]).

Various schemes have been devised and reported in the literature. Snyderman and Hunt [44] constrained their choice of encodable bigrams so that the initial characters belonged to a set of 8 "master" characters, selected primarily by frequency (see Pratt [29]), but arranged to include the vowels and the blank. The second characters belonged to a set of 21 "combining" characters, yielding a total of 168 encodable bigrams. All other characters were stored unaltered. Coding and decoding then depended in essence on the recognition of the "master" character.

Schieber and Thomas [33] placed no restrictions on the selection of bigrams (other than frequency), sacrificing the faster recognition of the previous scheme for an improved set of bigrams. A compaction of 43% was reported when 198 of the most common bigrams were encoded, compared to the 35% reported by Snyderman and Hunt [44].

Fouty [10] used bigram coding on databases of various languages and compared the performance to techniques based on variable-length coding (see 3.2). His results also indicate that it is possible to use a single set of bigrams for all the languages he considered (viz. English, German, French and Italian), and still achieve a compaction of about 40%.

In general, bigram substitution appears to be well suited for use on active files of consistent provenance. The compression and decompression routines are relatively fast and economical of memory. The main drawbacks are the selection of the bigrams to be encoded, which is usually based on statistics obtained from a sample of the file, and the order in which the bigram substitutions are to be made. Note that this method can be combined with a fixed substitution for a small number of common character strings longer than two characters (preferably starting with one of the encodable bigrams, for ease of programming).

3.1.3. Common Phrase Suppression

In this method a string of text is searched for repeating phrases (character strings) of any length. The phrases are then removed from the text and replaced by fixed-length reference numbers. The reference number is to an entry in a dictionary of phrases, contained in memory.

The two main problems associated with this method are firstly, the choice of a good set of phrases, and secondly, the use of the phrases in an order that will achieve best compression. These problems are illustrated in the following example. It should be noted that these two difficulties are common to many other compression techniques (see 2 above and 3.2 below), and various algorithms have been published to help minimise their effect (McCarthy [24], Wagner [47], Schuegraf and Heaps [36] and [37], Doucette [9], for example.)

Consider the input string "ABCXABCYABCZXABCY". If we choose the phrases "XABCY" and "ABC", and apply them in that order to the string, we get a compressed string of length 5 characters (assuming that the reference number is 1 character long). However, as "ABC" occurs 4 times, we could have chosen it first instead, thus obtaining a compressed string of 9 characters in length. Note that in general substituting for the longest phrase first does not necessarily give the best compression.

The overhead at compression time is relatively high as quite a bit of processing is needed both to find the optimal set of phrases to be suppressed (McCarthy), and to find the best substitution order (Schuegraf and Heaps [36] and [37] Doucette [9]. If the text is reasonably stable, however, the phrase set selection algorithm need only be run once. Storage savings of from 27% (Schuegraf and Heaps [37] to 58% (McCarthy [24]) have been reported, although the latter figure was obtained through combining this method with Huffman encoding of the phrase references and characters.

Note that the use of text and word fragments is very similar to this technique, but will be covered later because of their possible use as content indicators in automatic indexing.

3.1.4. Adaptive Character String Substitution

This is a more complex method than fixed substitution (3.1.3. above), with resultant higher overhead in time and space. The compression achieved is good and no preliminary generation of a code table is necessary. A brief description follows (Stoneburner).

The compressor starts with its code tables empty, except for one entry for each character in the input set. The input text is scanned and a count kept of the occurrence of each bigram. When a specific count reaches a threshold value, the compressor automatically defines a substitution code for that bigram. The definition is passed to the decoder as a special instruction in the compressed data. The process is iterative in the sense that counts are kept for the use of defined substitution codes in combination with other substitution codes or characters. Thus, although each substitution code is defined in terms of two other characters or substitution codes, it may represent a long string of characters in the original text. For example, a long string of X's in the input will result in the definition of a code for XX (say /), then the definition of a code for // (say $, which represents XXXX in the input), then a code for $$ (representing XXXXXXXX in the input), and so on. Obviously some large tables are needed and considerable time is spent searching and managing them. The decompressor only has to recognise a new substitution code definition and update its table accordingly. Decompression then consists of substituting the correct character string for each code in the compressed data.

On large files this technique results in a compression slightly better than a character-based Huffman code (Stoneburner). However, this depends on how "regular" the input stream is, and on how much of the file has been processed (due to "warm-up" or "learning"). It is thus suitable for files of several thousand words or more, and, due to the high overhead, inactive files are preferred.

3.2 Variable length coding for Characters and Character Strings

3.2.1 Huffman Codes

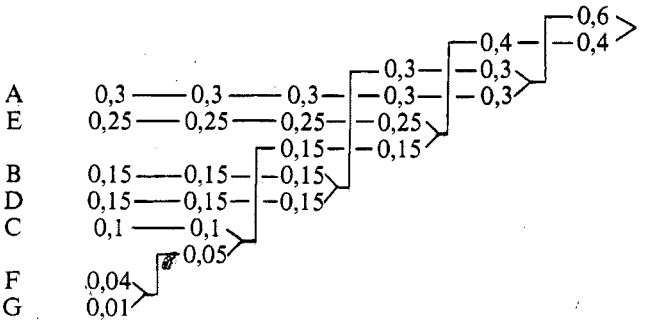
A procedure for producing a "minimum redundancy code" was described by Huffman [14], and minimises redundancy with respect to the element selected — normally the single character. All redundancy would be eliminated if the characters occured independently. Given this limitation, the procedure produces, for a finite set of characters with a given (fixed) frequency distribution, a set of codes with a minimum average length. In other words, the longest codeword is associated with the least probable character.

Additionally, when variable length codes are used, there must be a way to tell where one character ends and the next one starts. One of several ways of achieving this is to ensure that the code has the "prefix property", that is, no short code group is duplicated as the beginning of a longer group. Huffman codes have this prefix property.

The algorithm for generating a Huffman code is most easily illustrated by following an example. Assume a 7-character alphabet, with the following probability distribution:

A	B	C	D	E	F	G
0,3	0,15	0,1	0,15	0,25	0,04	0,01

The characters are first ordered by probability. A coding tree is then built up by, at each stage, connecting the two lowest probabilities to a node which is assigned a value equal to their sum, and then placing this new node appropriately in the ranking. The process continues until only one node remains (see following figure):



(Note that if the new node has a probability equal to another node in the list, the new node should be placed above the old node(s) which have equal probabilities. Schwartz [39] shows that this will minimise the codeword lengths).

The codes are then determined by assigning to each branch from a node a value of one or zero and reading from the root to each character. If, in the above figure, all upper branches are given a value "0" and all lower branches a value "1", the following set of codes results:

Character	A	E	B	C	D	F	G
Code	01	10	000	110	01	1110	1111
L_iP_i	0,6	0,5	0,45	0,3	0,45	0,16	0,04

Average length = ΣL_iP_i = 2,50 bits per character.

For decoding purposes, it is convenient to have codewords of the same length adjacent to each other in the tree, which leads to the following code (Schwartz and Kallick [40]):

A	E	B	C	D	F	G
00	01	100	110	101	1110	1111

Hoffman encoding works effectively on text and almost any other highly redundant data, usually achieving a compaction of about 50%. It has also been frequently used on business-type data files. When combined with other methods such as blank suppression (Katcher [15]), slightly better results are obtained.

It is desirable that the statistical properties of the file being compressed do not change over a period of time. Thus a new code table may be needed for a file if the character frequencies have changed considerably. In fact, as can be appreciated, the most important single factor in the development of a Huffman code for a file is the choice of the base character set (for instance, see Ruth and Kreutzer [31]). If a Huffman code is based on a subset of the possible characters, a copy code should also be provided. This is a special codeword which is used to indicate that the character following it is reproduced exactly as it occured in the source file.

It should also be noted that it is possible to construct non-optimal prefix codes which satisfy arbitrary constraints and yet produce an average code lengtrh close to the optimum. One such constraint was used by Gilbert and Moore [11], in which the numerical value of the codes maintains the ordering of the input symbols (which may not be in probability order). This is useful in sorting coded alphabetical data. Gilbert and Moore give an example of text coding using their algorithm, and report an increase in cost of only 1,9% over Huffman. Hu and Tucker [13] give an improved algorithm for the generation of such codes.

A practical disadvantage of all these minimum redundancy codes is that decoding must be done on a bit basis, usually by a time consuming tree search procedure.

3.2.2 Other Methods

In an attempt to speed up decoding and also to handle variable length codes on a fixed word length computer more efficiently, various other techniques have been proposed.

Bemer [2] assigned variable code lengths to words, and represented them by an integral number of bytes (and thus not optimum as in a Huffman code), the length of any code being indicated by the first bits of the code. This increased the coding speed by sacrificing some coding efficiency. A compression of 35% was achieved but a very large word dictionary had to be maintained. Heaps [12] gives a detailed description of a similar scheme, together with a thorough storage analysis, while particulars about the program design for coding and decoding are reported by Thiel and Heaps [46].

Mullarney [26] uses so-called optimal "self defining length codes". These are prefix codes with the additional property that the length of any code can be determined from an inspection of a fixed number (k) of the initial bits of the code. Decoding is then reduced to two shifting and indexing operations per symbol: a shift of k bits, and their use to address a table; a further shift of the indicated number of bits, and their use to retrieve the decoded character from the table. Mullarney quotes an average cost increase of less than 5% over Huffman codes.

A consequence of the greater efficiency of any variable length encoding is increased sensitivity to bit errors, resulting in loss of synchronisation or incorrect deconcatenation of codes. Prefix codes are self synchronising, and Mullarney has shown that, with a deliberate perturbation of the coded data, synchronism is typically regained within 5-15 characters.

3.3 Word Dictionary Techniques

3.3.1 Split Dictionary Encoding

By dividing the word dictionary into several distinct sections, long words may be synthesised, instead of having a separate entry for each word to be encoded, as occurs in an integrated dictionary. This technique may be thought of as a sophisticated extension of the non-adaptive character string substitution methods already mentioned.

In a stem and suffix system, there are separate dictionaries for word stems and suffixes. The input string is scanned and whenever the stem of a compound word is found in the stem dictionary, the suffix dictionary is searched to see if it contains the suffix of the compound word. If it does, the word is replaced by codes for the stem and suffix (care must be taken of words with multiple suffixes).

Schwartz [38] shows that using a split dictionary of stems and suffixes allows more words to be encoded for a given dictionary size than could be encoded with an integrated dictionary. Schwartz included a table of common words thus also taking advantage of the more compact and faster encoding of an integrated dictionary. In a later paper (Schwartz and Kleiboemer [41], results show that for a text length of 19 710 words, 79,49% were in the common word dictionary, 12,18% had to be synthesised and 8,33% were spelled.

Another similar method was described by White [49], in which frequent words and letter combinations (bigrams and trigrams) were held in a split dictionary. The encoder was designed to collect statistics of the input text and of the use of the dictionary, thus enabling some hand optimisation of the dictionary to be performed. The final dictionary contained 1 340 entries and produced a claimed reduction of 47%. Instead of using fixed length codes for the dictionary entries, it was calculated by White that Huffman codes would have produced a reduction of 4% more.

As these techniques do not have a dictionary entry for all possible words, a spelling mode is essential to enable words not listed to be spelled out character by character. If the dictionary can be fixed in advance, active files can be compressed by these methods.

3.3.2. Intermediate Dictionary Compression

This is basically a word dictionary technique and is due to Cullum [7]. It uses Huffman and run-length coding as integral parts of the method, and all words must appear in the dictionary. (The Huffman codes are defined algorithmically so that no code table is used).

The entire file is scanned to compile a complete dictionary of words and break characters (asterisk, period, comma, etc.). The break characters are then Huffman coded, based on their frequency count, while the words are "Huffman coded" assuming they are equiprobable (almost the same as assigning binary numbers to them), to save SPU time.

The file is then encoded as follows. A sub-string of say 1 500 words and break characters is taken from the file. A binary "presence vector" (of length equal to the number of words in the dictionary) is then constructed by turning the corresponding bits on if the word that that bit represents is present in the sub-string. This presence vector defines the intermediate dictionary (ID), which consists of all the words corresponding to the bits set to 1. The total number of words in the ID is counted and each word is assigned a "Huffman code" based on its position in the ID (see note on word coding above). The encoding for the string then consists of:

- The compressed presence vector in run-length encoded form
- The encoded word and break characters comprising the string in the order in which they appear. These are encoded by the concatenation of a 0 bit with the code for each break character or a 1 bit with the code for each word.

The main dictionary is included at the start of the compressed file and is followed by the codes for the break characters and then by the codes for all the sub-strings which together comprise the original file.

With the large overheads in this method it is suitable only for large inactive files. The files should also be subject to infrequent searches only, as searching must be done serially. Cullum reported a good compression of 53%, recorded on a source file with many short words and misspellings (both of which are unfavourable to this method).

3.4. Binary Data Compression

3.4.1. Run-Length Encoding

Some data tends to be in the form of sparse or low-density binary strings, that is, strings that have a preponderance of zero bits (or, similarly, of one bits). Such strings also arise in some of the compression techniques previously alluded to. A few of the many ways that have been devised to compress such strings will be briefly mentioned here.

The count of zero bits between two consecutive one bits may be encoded using a Fibonacci code (Kautz [16]). These codes are variable length binary codes representing the positive integers and which have the property that no code has a run of s consecutive ones, where s is an integer dependent on the code and chosen by the user. The codewords are then separated by the insertion of strings of s ones (This method is useful only for strings with less than 10% ones.).

An alternative way to encode the counts of runs of zeros is by exponent-fraction encoding. Each integer is encoded as an r-digit exponent (r is fixed for the code) followed by a "fraction" having a number of bits equal to the binary value of the exponent. The fraction then gives the count of zeros.

In a method termed "asynchronous compaction", the following transforms are applied to the original binary string:

00 -- 0 ; 01 -- 11 ; 1 -- 10

These reduce the number of zeros in the string, and are applied repeatedly until no further compaction results. Since the transform has a unique inverse, the original string can be reconstructed provided the number of times the transform was applied is known. (This could be supplied in a control field, along with the length of the compacted string). Long strings are compressed in sections. The method works well on strings that are not very sparse.

Lynch [19] biased the number of zeros in the source data by reassigning character codes ranked in order of the number of zero bits in a run to the characters ranked in order of frequency (thus the most frequent character is assigned the code with all zero bits). An extension of this technique assigns similarly ranked 2-byte codes to the bigrams ranked by frequency. Fixed 3-bit (and also 4-bit) codes are then used to represent runs of zeros (with one bits being encoded as runs of zeros of length zero). Reductions of the order of 30% were obtained using bigram biasing and 4-bit codes.

3.4.2. The COPAK Compressor

The COPAK alphanumeric compressor (de Maine et al [8]) is a recursive bit pattern recognition technique. It is fully automatic and stores all control information necessary for decompression with the compressed data. The input can be any arbitrary string of characters, numbers, codes or bits. Compression is achieved with two basic bit pattern recognition routines (Type I and Type II) which operates in either slow- or fast-mode.

In Type I compression, a codeword is substituted for a recurring bit pattern in the data string to be compressed. (A codeword is a character which does not appear in the input.) In Type II compression, codewords are removed from the string, and their locations indicated by bit-maps. A bit-map is a bit string with one bit for each character position in the input string. Each bit that is turned on discloses a position in the data string where the particular codeword is to be inserted during decompression. For example, the string "computer science" could be represented as "c(1000000000100010)omputer siene". If we omit trailing zeros, and also use a bit map for e, the string becomes "c(100000000010001)-e(0000010000101)omputr sin". In decoding, the substitution for e must precede that for c, since the bit-map for c has positions for e's in the string.

The control information stored with the compressed data string contains the codewords, the bit patterns they replace, and the bit-maps for the codewords if used. Thus decoding is accomplished by stepping backwards through the control information of the string.

In the slow-mode of compression, the input string is searched to determine the most frequently recurring bit patterns to be replaced by codewords. If these bit patterns are supplied to thye COPAK compressor by the user, this step is eliminated, giving fast-mode compression.

Although this technique does not require much storage, the processing time is considerable, making it suitable for static files only. Its performance is comparable to adaptive character string substitution, achieving compressions of around 50% on natural language texts.

3.5 Fragments

3.5.1. In Compression (See also 3.1.3)

Walker [48] used variable length character strings (which he called X-grams) to store names in a compressed form by concatenation of the codes for the X-grams. The set of X-grams, chosen on both frequency and length considerations, form a dictionary in which each entry is associated with a fixed length code.

Another proposal for the use of variable length character strings was made by Clare et al [5]. These string can be termed fragments since they are not limited to be elements of words, provided they are part of the record. (Word fragments on the other hand, refer to sub-strings of complete words only). Clare et al also placed another restriction on the choice of fragments, namely that they should occur with approximately equal frequencies in the file. The fragments are chosen such that they occur with a frequency below a certain threshold, and they are then used as the indexing, compression and retrieval elements.

Schuegraf [35] considers fragments such that there is no mutual overlap between the fragments, and which are chosen so that their frequency of occurrence is greater than or equal to a certain threshold (see also Schuegraf and Heaps [36] and [37], Doucette [9]. Schuegraf achieved a compression of about 60% on an issue of a MARC tape 51 047 characters long.

Further implications and the construction of equifrequent character strings are discussed by Lynch [20]. The frequency of occurrence of characters and fragments is adjusted so that high frequency characters are "absorbed" into lower frequency fragments. The symbol set so obtained results in a higher entropy (because of near equal probabilities of occurrence) and can be used to generate, or encode, the source file

3.5.2. In Indexing and File Design

In inverted file directories, fragments have been found to be reliable "surrogate words" (Schuegraf, Schipma [34] and they are also useful in file or bibliographic searhing (Colombo and Rush [6], Clare et al [5]. Lynch et al [21], Onderisin [28]. This is so particularly in files of consistent provenance; the fragments constitute the attributes which are employed singly, or in combination, to differentiate the items in the search file. (It has also been stated that fragments usually include or cover morphemes — Rickman and Gardner [30]. One could possibly consider fragments as computer equivalents of mnemonics, used for improving recoverability and discriminability (Norman and Bobrow [27]). Rickman and Gardner also showed that bigrams have a meaningful measure of association with index terms and can be used in automatic indexing. This was shown to hold for fragments as well by Barton et al [1] and Schuegraf

In addition, equifrequent fragments satisfy some of the conditions necessary for indexing terms to exhibit good discrimination (Salton and Wong [32].).

In inverted file design, the main advantages of using fragments instead of words is a reduction in the number of dictionary entries and the consequent saving of storage space. Schuegraf uses equifrequent fragments and, with the assumption that they are evenly distributed over all documents in the file, the inverted file directory consists of keys each with an approximately equal number of entries. Furthermore, the fragment list (or keys in the directory) is fixed even if the file increases in size (recall that the number of word types usually increases with an increase in the number of

tokens). This leads to considerable simplification in the design of inverted files, allowing for easier updating.

4. Conclusion

Algorithms have been discussed which have been used by various investigators to compress text files. Although compression figures have been stated, it must be emphasised that they depend heavily on the type of text being compressed and on the number of bits used in the computer to represent a character. Until a model is presented which enables a theoretical evaluation of the relative effectiveness of the many compression techniques to be made, the technique used in a particular application will still usually be chosen by experience and experiment.

Fragments appear to hold great promise for both compression and automatic indexing. However, many aspects of their use still require investigation: for example, fragment distribution and selection to maximise their discrimination value; coding algorithms with indexing and/or compression as the main objective; and the determination of a fragment set over different files.

Acknowledgement

This work was carried out while the author was visiting the Department of Computer Science at Cornell University, Ithaca, New York. It was partially supported by a bursary from the S.A. Council for Scientific and Industrial Research, which is gratefully acknowledged.

References

1. BARTON, I.J., CREASEY, S.E., LYNCH, M.F. and SNELL, M.J. (1974). An information-theoretic approach to text searching in direct access systems, *Comm ACM*, **17**, 345-350.
2. BEMER, R.W. (1960). Do it by numbers — digital shorthand, *Comm ACM*, **3**, 530-536.
3. BOOKSTEIN, A. and FOUTY, G. (1976). A mathematical model for estimating the effectiveness of bigram coding, *Infor Proc and Man.*, **12**, 111-116.
4. BURTON, N.G. and LICKLIDER, J.C.R. (1955). Long-range constraints in the statistical structure of printed English. *The Amer J. of Psych.*, **68** 650-653.
5. CLARE, A.C., COOK, E.M. and LYNCH, M.F. (1972). The identification of variable-length, equifrequent character strings in a natural language data-base, *The Comp J.*, **15**, 259-262.
6. COLOMBO, D.S. and RUSH, J.E (1969). Use of word fragments in computer based retrieval systems. *J. of Chem Doc.*, **9**, 47-50.
7. CULLUM, R.D. (1972). A method for the removal of redundancy in printed text, Ph. D dissertation, Dept. of Comp Sc., Univ. of Illinois.
8. DE MAINE, P.A.D., KLOSS, K. and MARRON, B.A. (1967). The SOLID System II. Part II Alphanumeric compression, U.S.

Nat Bur of Standards, Tech Note 413.

9. DOUCETTE, V.L. (1977). An algorithm to select a fragment dictionary for data base compression, to be published in J. of the ASIS
10. FOUTY, G. (1973). Techniques for the compaction of machine-readable bibliographic records, MA dissertation, Grad Lib. School, Univ of Chicago.
11. GILBERT, E.N. and MOORE, E.F. (1959). Variable length binary encodings, *Bell System Tech J.*, **38**, 913-967.
12. HEAPS, H.S. (1972). Storage analysis of a compression coding for document data bases, *Infor*, **10**, 47-61.
13. HU, T.C. and TUCKER, A.C. (1971). Optimal computer search trees and variable-length alphabetical codes, *Siam J on Appl. Math.*, **21**, 514-532.
14. HUFFMAN, D.A. (1952). A method for the construction of minimum-redundancy codes, *Proc of the IRE*, **40**, 1098-1101.
15. KATCHER, A.M. (1971). Efficient utilization of limited access archival storage in a time shared environment, *Proc of the Symp on Infor Stor and Retr*, April 1-2, 1971, New York, 197-205.
16. KAUTZ, W.H. (1965), Fibonacci codes for synchronisation control, *IEEE Trans on Info Theory*, **IT-11**, 284-292.
17. KUCERA, H. and FRANCIS, W.N. (1967). Computational Analysis of Present-day American English, Brown Univ. Press, Rhodes Island.
18. LESK, M. (1970). Compressed text storage, Computer Sc Tech Report No. 3, Bell Tel Labs, Murray Hill, N.J.
19. LYNCH, M.F. (1973). Compression of bibliographic files using an adaptation of run-length encoding. *Infor Stor and Retr.*, **9**, 207-214.
20. LYNCH, M.F. (1977). Variety generation — a reinterpretation of Shannon's mathematical theory of communication, and its impli cations for information science. *J. of the ASIS*, **28**, 19-25.
21. LYNCH, M.F., PETRIE, J.H. and SNELL, M.J. (1973). Analysis of the microstructure of titles in the INSPEC data-base. *Infor Stor and Retr.*, **9**, 331-337.
22. MANDELBROT, B. (1953). An informational theory of the statistical structure of language. In "Communication Theory", (Jackson, W., ed.) Academic Press, New York, 486-502.
23. MANDELBROT, B. (1961). On the theory of word frequencies and on related Markovian models of discourse, *Proceedings 12th Symposium in Applied Mathematics*, American Math Soc. 190-219.
24. McCARTHY, J.P. (1974). Automatic file compression, in *Int. Comp Symp*, 1973. North Holland, Amsterdam, 511-516.
25. MILLER, G.A., NEWMAN, E.B. and FRIEDMAN, E.A. (1958). Length-frequency statistics for written English, *Information and Control*, **1**, 370-389.
26. MULLARNEY, A. (1975). Text compression, Ph.D. dissertation, Dept. of Comp Sc., Univ of Dublin.
27. NORMAN, D.A. and BOBROW, D.G. (1976). Recoverability and discriminability : factors in memory retrieval, Personal communication.
28. ONDERISIN, E.M. (1971). The least common bigram: A dictionary technique for computerized natural-language text searching, *Proc. ACM Annual Conf.*, 82-96.
29. PRATT, F. (1939). Secret and Urgent : the Story of Codes and Ciphers, The Bobs Merril Co., New York.
30. RICKMAN, J.T. and GARDNER, H.W. (1973). On-line index term predictions using bigram-term associations, *Proc. ACM Annual Conf.*, 262-270.
31. RUTH, S.A. and KREUTZER, P.J. (1972). Data compression for large business files, *Datamation*, **18**, 62-66.
32. SALTON, G. and WONG, A. (1976). On the role of words and phrases in automatic text analysis, *Comp and the Humanities*, **10**, 69-87.
33. SCHIEBER, W.D. and THOMAS, G.W. (1971). An algorithm for compaction of alphanumeric data, *J. of Lib Auto.*, **4**, 198-206.
34. SCHIPMA, P.B. (1971). Term fragment analysis for inversion of large files, *IIT Research Institute Research Report*, 1-16.
35. SCHUEGRAF, E. J. (1974). The use of equifrequent fragments in retrospective retrieval systems, Ph.D. dissertation, Dept. of Computing Sc., Univ. of Alberta.
36. SCHUEGRAF, E.J. and HEAPS, H.S. (1973). Selection of equifrequent word fragments for information retrieval, *Infor Stor and Retr.*, **9**, 697-711.
37. SCHUEGRAF, E.J. and HEAPS, H.S. (1974). A comparison of algorithms for data base compression by use of fragments as language elements, *Infor Stor and Retr.*, **10**, 309-319.
38. SCHWARTZ, E.S. (1963). A dictionary for minimum redundancy coding, *J. of the ACM*, **10**, 413-439.
39. SCHWARTZ, E. (1964). An optimum encoding with minimum longest code and total number of digits, *Infor and Control*, **7**, 37-44.
40. SCHWARTZ, E.S. and KALLICK, B. (1964). Generating a canonical prefix encoding, *Comm ACM*, **7**, 166-167.
41. SCHWARTZ, E.S. and KLEINBOEMER, A.J. (1967). A language element for compression coding, *Infor and Control*, **10**, 315-333.
42. SHANNON, C.E. (1949). A mathematical theory of communication, in "The Mathematical Theory of Communication" by Shannon, C.E. and Weaver, W., The University of Illinois Press, Urbana, 3-91.
43. SHANNON, C.E. (1951). Prediction and entropy of printed English, *Bell System Tech. J.*, **30**, 50-64.
44. SNYDERMAN, M. and HUNT, B. (1970). The myriad virtues of text compression, *Datamation*, **16**, 36-40.
45. STONEBURNER, P. (1977). Compression techniques. Personal communication.
46. THIEL, L.H. and HEAPS, H.S. (1972). Program design for retrospective searches on large data bases, *Infor Stor and Retr.*, **8**, 1-20.
47. WAGNER, R.A. (1973). Common phrases and minimum-space text storage, *Comm ACM*, **16**, 148 152 and 183-185.
48. WALKER, V.R. (1969). Compaction of names by X-grams, *Proc. of the Amer Soc for Inf Sc.*, **6**, 129-135.
49. WHITE, H.E. (1967). Printed English compression by dictionary encoding, *Proc IEEE*, **55**, 390-396.
50. ZIPF, G.K. (1949). Human Behaviour and the Principle of Least Effort, Addison-Wesley, Massachusetts.

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 in double-spaced typing on one side only of 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 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

Contents/Inhoud

Spectrum Offset