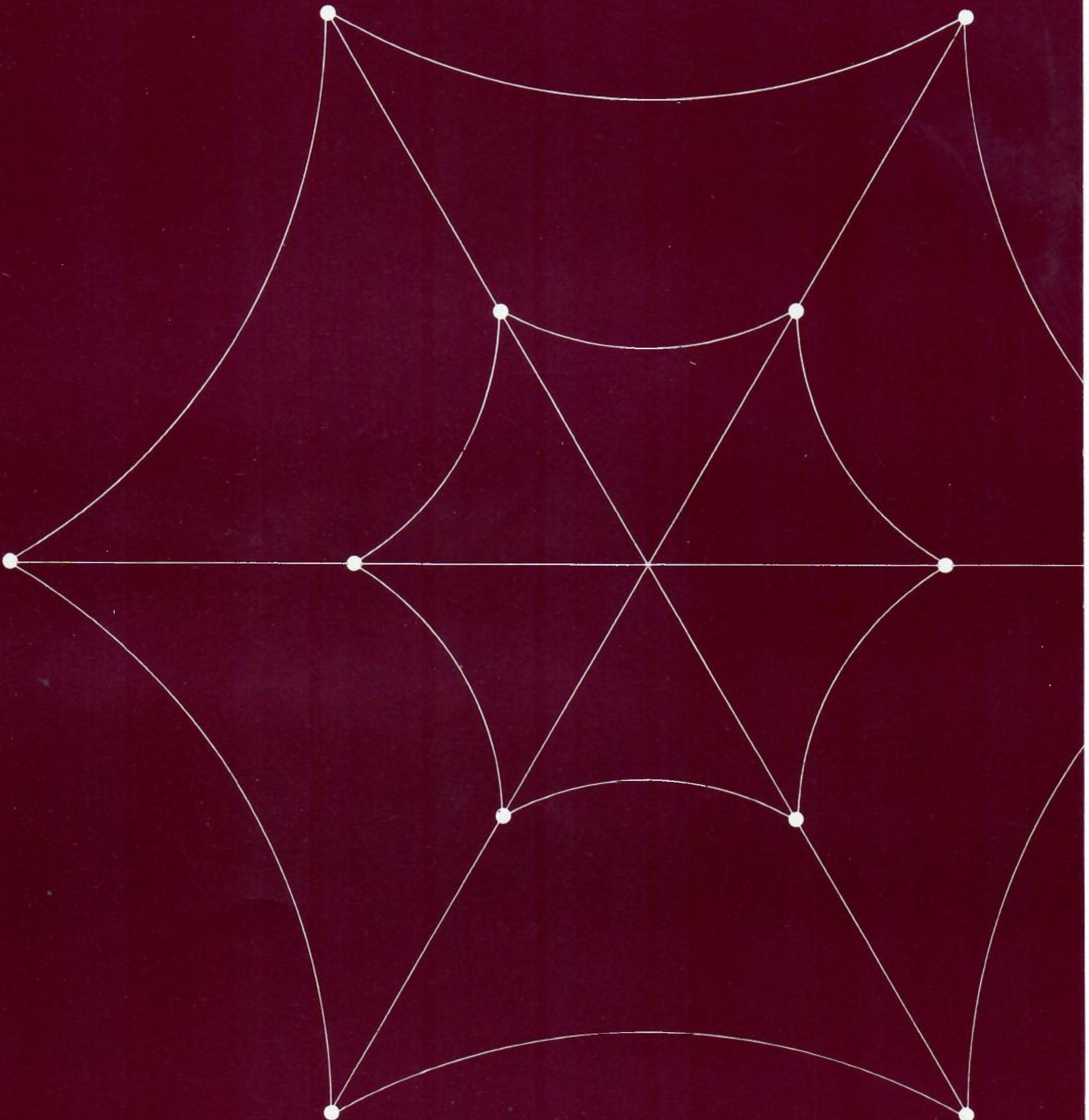
DISTRIBUTED COMPUTING

a review for industry

a conference sponsored by the
Science and Engineering Research Council

3-4 March 1983
National Computing Centre
Manchester



DISTRIBUTED COMPUTING

A Review for Industry

Contents

Each Section contains a review report and viewgraph material which will be used in the conference presentations.

"RINGS AND THINGS"
A Report for SERC/DCS Committee on Local Area Networks (or Local Networks),

by E. B. Spratt University of Kent

1. Preamble

This report has been prepared on behalf of the SERC for the Industrial Distributed Computing Systems Conference in March 1983. An attempt has been made to cover the main types and applications of Local Networks in use within the U.K. with reference to the position in the last quarter of 1982 together with some discussion on comparative assessments and standards. The emphasis is on systems in use within the Academic Community i.e. sites within the SERC (Science Engineering Research Council) the Universities and Polytechnics.

The views presented in what follows are personal to the author and do not in any way reflect the policy of the SERC, the University of Kent or indeed any other official body with which the author is associated. The task of selecting suitable material from the large (and growing) amount of available information has not been easy and any omissions should be viewed in this light.

Numbers in square brackets e.g. [9], refer to individual items in the list of references given at the end of this report.

2. Organisation of the report

The material in this report is organised as follows

3. General Introduction

We commence by explaining what is meant by a local network. They have three distinctive properties.

(1) A diameter of up to 2 to 3 kilometres.

(2) A raw (or total) data rate exceeding one megabit per second.

(3) Owned by a single organisation.

The reader who is interested in a general introduction to local networks should consult [12] or [1].

There are two main reasons why local networks are required, and these are essentially the same reasons that organisations are interested in networks in general.

The first reason for an interest in local networks is to exploit the advantages of functionally distributed computing. Typically, if this approach is followed some of the computers are dedicated to specific functions such as terminal handling, data base management, file storage, printing or controlling industrial control equipment. This is the main relevance of local networks for the SERC Distributed Computer Systems research program. As is well known SERC made a policy decision some three years ago to standardise on one particular local network, namely the Cambridge Ring (which we consider further in a later section) to act as a common research vehicle for research groups in this area.

The second reason for the importance of local networks is to interconnect computers, terminals, and peripherals which are located either in the same building or in many cases in nearby buildings, in such a way as to enable them to intercommunicate and also to allow them all to access a remote host computer or another network. The presence of the local network makes it possible for the remote facility to connect to the local network at one particular point (which is usually called a gateway).

It is this latter reason which has led many Universities and Polytechnics to plan and in some cases to implement computer services based on local networks. The Joint Network Team of the Computer Board and the Science Engineering Research Council is finding and co-ordinating development work in this area. It will be recalled that the Computer Board is responsible for the provision of computer systems for central university computer services, whilst the Joint Network Team is funded jointly by the Computer Board and the SERC, to be responsible for co-ordinating network activities between the SERC and the universities and other sites within the Academic Community. One important aspect of this work is the implementation of standards and reference is made to this later in this report.

Local Networks are an important component of the SERC Common Base Software Policy which concentrates on two languages, Fortran 77 and Pascal on GEC and Prime minicomputers together with ICL PERQ workstations, where the latter are connected by means of Cambridge Rings.

Local networks differ from the wide area (or long haul) type in several ways (Fig 1). A crucial difference is that the developers of wide area networks are often compelled by legal or economic reasons to use the public telephone network, which is not necessarily the most suitable technical choice. On the other hand the designers of local networks can choose and even lay their own choice of high bandwidth cables. This gives them definite advantages. In particular, bandwidth (or simply the carrying capacity of the network) is not such a scarce resource as for wide area networks and thus it is not necessary to optimize its use with local networks to anything like the extent that is done in the wide area case.

4. A note on terminology

Here we explain certain terms which will be used in this report.

A local network (or indeed a wide area network) essentially consists of nodes which are linked together in some way by means of a transmission medium, these nodes usually consist of several logically distinct components which may or may not be physically distinct. For instance there is always a means of allowing an attachment to the transmission medium, some standard logic and device specific logic which is used to attach the actual user device. The latter could for example consist of a simple micro to operate sensory devices, a mini or even a mainframe computer (Fig 2).

One of the most crucial concepts in networking is that of a protocol. A protocol is a precise set of rules which enable computer systems to intercommunicate. It is important to realise that protocols are necessary with both the hardware and software aspects of communication and there is usually more than one level of protocol involved.

5. Technologies for Local Networks

In this section we consider a number of topics concerned with technologies for local networks. Although it may be fashionable to debate the issues in this area they seldom take the same importance to the end user.

5.1. Media for local networks

There are four areas to consider

- type of cabling
- broadband and baseband
- passive
- network layout

Before discussing these separately it should be noted that we assume serial transmission is being used. Although parallel transmission is feasible there appears to be no particular requirement for this at the present time due to the high speeds which are available for serial transmission and the costs of working with parallel transmission.

<u>Cabling</u>

The three main cable types are twisted pair, coaxial and fibre optic. There are many different forms of both twisted pairs and coaxial cable to meet various specifications, e.g. noise immunity, and choice of bandwidth. The important point is that both these choices are quite satisfactory for the majority of local network environments. The choice of cable type is usually determined by engineering considerations and installation issues. With environments which have high levels of electromagnetic radiation or there is a requirement for high speed serial transmission the natural choice is for fibre optic cable and there is little doubt this will come into much more common use after a year or so.

Baseband and Broadband

The difference between these two types of media is that in the baseband case there is a single channel for the information flow whereas in the broadband case there are a number of channels superimposed on a single cable using frequency division multiplexing techniques. A number of techniques are used to control access to the baseband media and these are considered in a later section. It should be noted that these techniques may also be used with individual broadband channels.

Broadband cabling is based on CATV (Cable Television) technology and has more bandwidth than a similar baseband system. However the broadband channels are independent, so it is necessary for attached devices to select channels using frequency agile modems, with special switching equipment being used to enable connections between different channels to be made. One issue which has to be faced with this technique is that modems operating at radio frequencies are required to access these channels and if the costs of such equipment are to be kept within reasonable bounds then the bandwidth of the individual channels must be roughly in the range 5 to 10 megabits/second. If this remains a serious limitation in the future then it would appear that the next generation of baseband systems which should work in the 50 to 100 megabit range would be very attractive. Although there is much debate on the broadband v baseband issue it is perhaps likely that the two media will co-exist as, to an extent, they are complementary.

Passive or Active Systems

There has been much discussion on the advantages and disadvantages of these systems. Two comments are perhaps in order, firstly, high reliability systems can be achieved at reasonable costs and secondly, the increasing availability of LSI components for devices such as repeaters should also significantly improve reliability. In either case it is essential to be able to rapidly locate and fix faults.

Network layout

There are two related issues here. Firstly the network topography, which is the way in which the cabling is actually installed, and secondly the network topology which is a way of describing the logical links between the nodes.

Various topologies have been proposed e.g. rings, busses and stars. However many actual networks are made up of more complex topologies. For instance it is possible to use a central ring with linked subrings and some systems consist of interconnected busses forming tree configurations. (See Fig 3). Major considerations when choosing a topology are network management, reliability and maintenance. The topography is of course directly related to the wiring costs and hence the economics of a local network.

5.2. Slotted Rings

This section refers to the specific case of a Cambridge Ring rather than a general slotted ring since the former has most of the important characteristics of this particular type of local network. The ring is formed from nodes which are joined by means of suitable cabling, this can be either twisted pairs or fibre optics and it is perfectly feasible to mix these two types. With twisted pairs the length of cabling between nodes is normally about 200 metres and there must be a repeater at each node to regenerate the signal. Some nodes may consist solely of repeaters but others may have user devices attached to them. In the latter case the repeater is used for the attachment to the transmission medium as noted in an earlier section. In this case the interface between the user device and the repeater is called a station, and such a node is often referred to simply as a station (Fig 4). The operation of the Cambridge Ring is based on the setting up and subsequent use of circulating slots or fixed size packets (or units of information) which pass continuously round the ring in a undirectional manner. Clearly there must be at least one of these packets. Each packet contains data, the address of the source and destination nodes together with control information.

This slot structure is created when the ring is turned on, using a special station called the monitor. Thereafter the monitor continuously checks this structure and corrects it if necessary. For example, there is a mechanism to set a bit to show as empty packets which may have become full due to errors.

The original packet size was 38 bits but the current version (known as CR82) includes two additional control bits that may be used in the implementation of higher level protocols. Each packet contains two 8 bit fields which are for the destination and source addresses and a 16 bit field for data. The remaining bits are used for control purposes, e.g. to indicate whether the slot is full or empty and whether the data was accepted (or rejected) at the destination (see Fig 5).

The stations each have a select register which can be set to either accept (or reject) all packets which are addressed to it or receive from a specified source. This provides a selection mechanism which, together with the values of the slot response bits provides efficient low level acknowledgement and control facilities.

It is not possible for any of the network nodes to monopolise the traffic since the sender clears its own packet after it has been round the ring and is not then allowed to use it again immediately. There is also a facility to prevent a node which is transmitting at a high data rate from swamping a node with a slow receiver. The above description assumes that all the nodes have devices attached to them, in actual rings some of the nodes may just be repeaters. The reader who wishes to refer to a further treatment should consult [5].

5.3. Token passing

In this type of network access to the transmission medium is by means of a token which is passed from station to station according to some set of rules set up by the network designer. A station can only transmit when it is in possession of the token. When a node has finished transmitting it passes on the token to the next node in a sequence which places all the nodes of the network onto a logical ring, though the medium itself need not be a ring. As with slotted rings it is necessary to provide suitable functions for initialisation, error recovery and monitoring the logical ring. In particular it is crucial to have procedures for restoring the token if it becomes corrupted.

With token networks the packets can be of variable length and high line utilisations are possible (see papers by Bux in [3]), but it is necessary to provide complete packet buffers.

Work is currently being carried out on the use of tokens on bus topologies and this is still at an early stage of development. In contrast many ring networks have been and are based upon token passing. An interesting recent example is the IBM experimental token ring which is described in [4] in a paper by Bux and his co-authors. In this ring each station is a potential monitor, though of course there is only one of these in actual use at any given time.

5.4. Carrier sense multiple access

The so-called Carrier Sense Multiple Access (CSMA) technique is used in a number of networks including the Xerox Ethernet [2, 10]. It is an example of a broadcast media access method. In broadcast systems each connected device is at liberty to broadcast its information over the network. Since two transmissions occurring at the same time will result in the data being corrupted it is necessary to have arrangements to overcome this problem.

In CSMA each node of the network "listens" to see if any other nodes are transmitting. If this is the case then the node defers its transmission to a later time. If however the channel is free then it begins its transmission. However, since signals take a short time to travel along the network, it is possible for two nodes to transmit at almost identical times thus causing a collision. Therefore transmitting stations listen to the first part of the transmission. A collision will be detected and each station will then stop transmitting. They then wait for random time intervals and retransmit. Due to these random time intervals it is unlikely that the transmissions from these nodes will occur at the same time again. This is called collision detection and the usual abbreviation for this type of broadcast technique is CSMA/CD.

This basic technique may be implemented in various ways but it is usually done with coaxial cable where the single inner conductor is monitored for the presence or absence of signals.

The rules of the CSMA/CD technique imply that this type of network is probabilistic, i.e. there is no absolute guarantee that a transmission can be completed within a given time or that a station can transmit at a guaranteed minimum data rate. In practice this does not appear to be very restrictive and data communication response times are usually rapid.

The information on an Ethernet is transmitted in a packet, the format of which is given in Fig 6.

6. Comparative Issues

Comparisons are notoriously difficult and local networks are not an exception to this rule. However some trends appear to be coming clearer and we will now consider them.

The costs involved in local networks often only form part of a total system cost and may in fact perhaps be less than 15%. One factor which is going to reduce costs and improve reliability is the availability of LSI components and these seem likely to become available in quantity during 1983, and they will then be speedily incorporated into products.

There are likely to be continual discussions about the relative merits of baseband and broadband systems. Each have their advantages, for example individual broadband channel speeds are currently limited in speed to some 10 to 12 megabits but there can be several of them. Much will depend upon developments in Frequency Agile Modems, which are used to attach devices to these channels. Another factor of importance in the U.K. is that there is less experience compared with the U.S. with the cable television technology (CATV) which is used with broadband networks, but voice and television transmissions can be carried out on a broadband network. On the other hand baseband speeds will probably increase past the 50/100 megabit range for rings using fibre optic cabling. A comparison between rings and other types of local network is given in [17].

Some of the points which are important when comparing networks are rather more down to earth and practical than some of the considerations above. Engineering and maintenance considerations are crucial in our view. From time to time things will go wrong and then faults will have to be speedily rectified. A lightning strike is one example of a phenomenon which has affected both ring and ethernet networks (with complete impartiality). Equally monitoring must be considered and embedded into the system in a suitable manner. Yet another important factor is that there is, as yet, only a small amount of feedback from customer sites.

These considerations lead one to the conclusion that it is too soon to start picking out outright winners in the local network field. Anyone contemplating the purchase of one of these systems would be well advised to avoid too many preconceived notions and to prepare a detailed specification of their requirements before starting detailed discussions with possible suppliers.

7. Protocols for local and wide area networks

In this section we present some information about protocols, since these are a vital component in any local or wide area network. A protocol, standard, is necessary for the orderly exchange of information between computer processes. Without standard protocols it is impossible for systems from different suppliers or manufacturers to intercommunicate in an effective manner.

Most authorities now agree that these issues should be discussed within the framework of the Reference Model for Open Systems Interconnection which was put forward as a recommendation by the International Standards Organisation in 1978 (known as the ISO/OSI Reference Model), see [18]. Although it was originally intended for Wide Area Networks, many of the ideas apply also to local networks, see [7].

Essentially the proposal states that the communication issue should be addressed in terms of a seven-layer model as shown in Fig 7. Each of these layers provides a certain subject of services to the overall set of network functions which are required. In general each layer provides facilities to the modules (software or firmware) in the layers above. The physical link layer transfers the information as a stream of bits.

The data link layer structures this bitstream in order to provide an error-free communication path between two nodes. The network control layer sets up the path between nodes, routes, messages, errors, intervening nodes, addresses messages and controls the flow of messages between nodes.

The transport layer provides the end-to-end control of the communication session once the path has been established, allowing processes to exchange data reliably and sequentially. This is independent of which systems are communicating or their location in the network. The two layers above this are concerned with providing facilities for the applications layer, which provides services which support the actual user tasks. File and job transfer, terminal protocols and things like Electronic Mail are dealt with at about this conceptual level.

There are several important matters arising out of this model. Local network protocols do not conform exactly to the first three layers as discussed above. However it is generally agreed that any interconnections or gateways between local and wide area networks should be at the Transport level which enables the essential end to control issue to be resolved. In this connection it will be recalled that the British Telecom PSS (or Switchstream 1) service essentially uses the first three layers in the form of the CCITT X25 standard protocol. Provided the Transport level has been suitably implemented on the Local Network it is then feasible to use the same higher level protocols (or layers) on both local and the wide area networks. Whilst this may not be appropriate for all applications; and in particular for certain types of distributed computer system, nevertheless it does provide the means to effectively interconnect wide area and local networks. So that for example files may be transferred between the two systems.

The Academic community in the U.K. has standardised on high level protocols (i.e. protocols above level 3) for wide area networks pending developments in International Standards, called the Coloured Book protocols. Namely

Yellow Book,	Transport Layer.
Blue Book,	File Transfer.
Red Book,	Job Transfer and Manipulation.
Green Book,	TS29 Terminal Protocol.
Gray Book,	Electronic Mail.

These are gradually being introduced for interconnection between community sites using either the Science Engineering Council Network (SERCNET) or the British Telecom PSS/Switchstream 1 service. Some of these sites, for example the University of Kent, have Cambridge Ring local networks and the coloured book protocols will also be used on them. It seems likely that the same developments will take place for Ethernets or other local networks on community sites.

However below the transport level the protocols for Cambridge Rings or Ethernets differ from the ISO/OSI model, there are several reasons for this. In particular the transient error rates are much lower (by three or four orders of magnitude) than in a typical wide area network, so with the higher speeds which are available in the local network case it is practical to correct these at a higher level rather than level two. Also addressing problems tend to be much simpler. Some details of the Cambridge Ring protocols at these levels are given in [5 and 14].

8. Standards for Local Networks in the U.K.

Two of the most important standard developments affecting the U.K. are CR82 (see [13]) for Cambridge Rings and the forthcoming U.S. IEEE802/DIX/ECMA standard for Ethernets. Though at the present time both of these have de-facto rather than genuine standard status.

As its name implies CR82 emerged in 1982 as a result of discussions between the four U.K. suppliers of Cambridge Ring components namely Logica-VTS, Orbis/Acorn, SEEL, Toltec, together with the Science Engineering Research Council and the Joint Network Team. The latter organisations acting on behalf of the academic community. There were two consequences arising from this development. Firstly, there was a change from a 38 bit to a 40 bit minipacket by adding 2 additional control bits (Fig 5) and any equipment supplied to the CR82 specification from one of the firms above will interwork with CR82 equipment obtained from any of the others. Whilst CR82 has not been approved by the British Standards Institute (or BSI) nevertheless it is a significant step forward.

It seems unfortunate that it did not prove possible to get CR82 considered by the European Computer Manufacturers Association (ECMA) or the IEEE802 Local Network Standard Committee set up by the Institute of Electrical and Electronic Engineers in the U.S.

The CR82 definition covers only hardware. Specifications for protocol layers above this have been prepared by working groups set up by the JNT, as discussed in a previous section. Details of this are given in [14].

Although Cambridge University and Ferranti collaborated over the design of an LSI version of the principal Cambridge Ring components to a slightly enhanced version of the CR82 standard, using Uncommitted Logic Array techniques, this has not come into widespread use. It seems likely to be superseded during 1983 by an LSI version commissioned by the SERC. There are of course significant advantages for everyone once the components are engineered in silicon. For example reliability is superior and the standard status is much clearer.

The Ethernet position is rather different, the original design was carried out by Metcalfe and Boggs [16] of the Xerox Corporation at Palo Alto Research Centre (Xerox Parc) as part of a project work on Offices of the Future. Subsequently the specification of a revised version was agreed with the Digital Equipment Corporation (DEC) and Intel, and this was published in August 1980. Each partner in this grouping brought distinct advantages to the exercise, for example, Intel was to develop an LSI version of the Ethernet components whilst DEC had expertise in systems, micro mini computers and mainframes. DEC, Intel and Xerox are referred to as the DIX group in what follows. This specification was widely circulated, and it was possible for other firms to obtain a manufacturing licence for a reasonable fee. The hope of the DIX group was that other firms would use the specification to build compatible equipment and to a large extent this is what happened.

The original DIX standard (see [10]) did not cover any software protocol layers, however Xerox have themselves defined such facilities and the ECMA input to IEEE802 has also resulted in a proposal for the higher protocol levels. These are not compatible with the Rainbow protocols, in particular the Transport level is somewhat wider in scope than the comparable Yellow Book Transport Service. Nevertheless the difference between these two levels is not large and it would be feasible to consider using the higher level Rainbow protocols on both Cambridge Rings and Ethernets. It is of course likely that many users of these networks will wish to define their own protocols for specific applications, particularly if the corresponding local networks are closed, i.e. they are entirely self contained and do not communicate with other networks.

In parallel with the DIX developments the IEEE in the U.S. set up a Local Network specification group know as the IEEE802 Committee and the DIX Ethernet specification was submitted to this body.

However the DIX Ethernet specification did not emerge as a recommendation of the IEEE802 committee, though considerable discussion ensued. In the meantime ECMA was considering Local Networks and came down in favour of a similar system which was backed by a large number of firms in the computer industry including ICL and CTL. ECMA then entered into discussions with IEEE802 and the DIX group. Late in 1982 all three bodies agreed upon a final version of the specification for submission to the IEEE. LSI versions have been developed by several suppliers and should be available in quantity during 1983. The reader should note that the IEEE802 work deals with other types of Local Network, specifically token rings and busses.

Although there are many Local Network products being marketed in the U.K. it is our view that careful attention should be paid to these two "standards". We have excluded discussion on local networks running at speeds lower than 10 megabits per second but it should be noted that there are several interesting products, e.g. the Acorn Econet which is a simple type of Ethernet, but currently only appears to connect Acorn systems. Another example is the Clearway system which is marketed by Real Time Systems Ltd, which may be used to interconnect equipment from different suppliers. It is a simple, effective and low cost system.

The reader who is interested in developments in U.K. standards and local networks should consult the recent report to the Focus Committee on Information Technology standards which is available from the Department of Industry. This contains recommendations on future work and in particular considers Electronic Telephone exchanges and their relation to local networks [11].

9. Current work in the U.K.

A substantial amount of the local network research and development effort in the U.K. is going into Rings. For example, research on high speed slotted rings is being carried out at Cambridge. Several Universities (Cambridge, Loughborough, University College London) are cooperating with British Telecom, Logica, GEC-Marconi on the Universe project. This is studying the possibilities opened up by interconnecting Cambridge Rings by means of a communications satellite using 1 megabit per second links, supplemented by normal terrestrial network connections. A general description of the status of this work during early 1982 is given by Kirstein et al in [8]. This reference also contains four specialist papers on the Universe project dealing with protocol architecture, encryption, network measurement and authentication.

There are a number of projects being carried out in the SERC Distributed Computing Programme all using the Cambridge Ring as a standard research vehicle, these include

Needham, R.M.	Cambridge	Developments of the Cambridge Ring
K.H. Bennett	Keele	A Distributed Filestore
H. Brown, S.E. Binns and D.J. Caul	Kent	Typesetting and Text Processing Servers for the Cambridge Ring
P.J. Brown and P.H. Welch	Kent	Compiling Servers for the Cambridge Ring
B. Randell	Newcastle	Reliability and Integrity of Distributed Computing Systems
C.A.R. Hoare, Stoy, J.E. and Harper M.K.	Oxford	Distributed Computing Software
R. Bornat	Queen Mary College, London	Pascal-M: A language for the design of Loosely-Coupled Computer Systems
D. Hutchinson and W.D. Shepherd	Strathclyde	Direct Comparison of Ring and Ethernet Type Systems. Gateways for the Interconnection of Cambridge Rings and Ethernet-like networks.
J.W. Hughes and M.S. Powell	UMIST	Multiprocessor Software Engineering
P.T. Kirstein	University College, London	Communication Protocols in the context of X25 Computer networks
I.C. Wand	York	Operating Systems for a Network of Personal Computers

Further details are given in the Annual Reports from the SERC on Distributed Computing Systems.

The University of Strathclyde has an SERC Research grant for a comparative study on Cambridge Ring and Ethernet Type local networks. This work will also involve the developments of a gateway between these two networks. This is worth noting in view of the likely importance of these two technologies in the U.K. and the fact that some organisations may be involved with both of them. One University carrying out work on broadband systems is Sussex, and this includes the development of frequency agile modems and an Ethernet channel.

Kent, Leeds and Oxford are working on development contracts from the JNT on Cambridge Rings. Kent are tackling terminal concentrators, ring to ring bridges, reconfiguration and monitoring. The work at Leeds and Oxford involves different types of fibre optic cabling, whilst Oxford are also working on printer servers and micro computer interfaces. Other JNT projects on Ring interfaces involving both hardware and software, for several different computer systems including Prime, DEC Vax (Unix and VMS) and GEC4000, are being carried out on collaborative projects involving Universities, Polytechnics and Industry. It is interesting to note that these interfaces will operate at the Transport level. Other developments have been carried out at the Edinburgh Regional Computer Centre which include improved version of ring components and micro and minicomputer interfaces. It is anticipated that most of the items described in this paragraph will be marketed.

10. Local networks which are available in the U.K.

There are four firms currently active in the Cambridge Ring field, Logica VTS, Orbis, SEEL and Toltec. Logica supplied the original SERC Rings and has just announced a fibre optic ring cabling product, and has also developed a fileserver. SEEL are working on fibre optics and ways of increasing reliability by duplicating certain components. It will be recalled that these firms all supply ring components to the CR82 standard.

Some twenty companies have made a commitment to the ECMA "Ethernet" local area standards and since these will eventually be compatible with the IEEE802 and DIX standards, there is likely to be considerable effort put in by the relevant U.K. companies who include ICL, Computer Technology Ltd (CTL, NTL and OTL) and this implies work on both baseband and broadband systems.

An interesting U.K. developed local network system is available from Xionics Ltd. Their system consists of two networks in one - there is the Xinet ring network to which minis, mainframes and word processors are connected using so-called intelligent sockets. The network allows for 4095 addresses using 256 byte packets. With 16 bytes reserved for error checking, addressing, and flag data. Hogging is avoided by letting each intelligent socket have a packet of its own.

The Xinet is attached to the Xibus which is a closely coupled network of processors which manage disc stores and carry out monitoring functions (as in the Cambridge Ring). There is extensive duplication within the Xionics system as regards cabling, power supplies and discs, to provide a high degree of redundancy.

Several networking type interfaces are available and others are planned, e.g. Telex, Ceefax, Oracle, the interconnection of Xionics systems, PSS and mainframes from suppliers such as IBM, ICL and DEC. The standards position is not entirely clear with this network and it appears to be a closed system with gateways providing any required compatibility with other systems.

The general picture which emerges is that of a rather limited number of installations during 1982, which is likely to increase enormously in 1983.

11. Some current applications of networks

Local networks are being used in the academic community in at least two different ways. Many sites are using them as research vehicles for work on Distributed Computing, whilst others are using them as the basis for the provision of a computer service. An example of the former is the University of Cambridge who also provide a departmental computing service. A detailed description of this system which is based upon the Cambridge Ring, is given in [5]. Perhaps the main characteristic of this work is that users of the system access banks of processors attached to the network from terminals relying on a shared filestore (or fileserver) on a separate node. There is no local filestore directly attached to the terminal or the processors. On the other hand the University of Kent based their computer service on a Cambridge Ring network service in January 1980. This has enabled a considerable rationalisation in the communication facilities. (Spratt [8]) describes the current state of this system and (Spratt [3]) contains an account of earlier work. Another example of a University which has based part of its computer service on this type of network is the Regional Computer Centre at Edinburgh. University College, London is using Cambridge Rings both in their research work and in their provision of gateway facilities to ARPANET for SERC Research Workers.

Very few Ethernet systems appear to be in actual use within Universities or Polytechnics during the last quarter of 1982. Apart from the work at Strathclyde mentioned earlier. However it is clear that this situation will change in 1983 as firm plans are known to exist at a number of sites.

The situation in the non-academic community is less clear. In general it appears that a number of baseband and broadband networks have been installed by U.K. firms essentially based on imported product mainly from the U.S., these systems often provide Ethernet facilities. Examples of this are Thame Systems and Case, who are using Ungermann Bass Net One equipment, and Network Technology Ltd who are using Sytek systems. A number of these systems are being installed under a government sponsored programme concerned with Office Automation. Some Ethernets have been installed by Xerox, in particular for the Greater London Council. Xionics have installed a number of systems including one in the Cabinet Office.

The four CR82 Cambridge Ring suppliers each have an installed user base, with Logica taking a leading role in respect of customers outside the U.K. Other firms have developed other types of ring, e.g. Racal with their Planet system which was announced during 1982.

12. Networks Users Association Survey

This is basically a U.S. organisation and some of the results of a recent survey on local networks as reported in the Localnetter Newsletter (see References-reports) are summarised below.

Q1: Responding Organisation Classification.

Miscellaneous End Users	41.5
Banking	18.4
Government	16.1
Education	12.0
Manufacturing	12.0

Q2: Application of networks under following four categories

Host Access	37.6
Office Automation	32.8
Intra Data Centre	19.8
Manufacturing	9.8

Q3: Transmission preference under following four categories

Broadband	37.7
Baseband	27.9
PBX	19.6
Ring/Loop	14.7

Note. This is a rather odd question, presumably the Ring/Loop plus the Baseband figures of 42.6 is the important point to note.

Q4: Media Access Technique preference

CSMA/CD	32.9
Don´t Care	30.5
Token	20.2
Don´t know	16.4

Q5: Type of Access required

Statistical (Contention)	40.3
Don´t care	37.0
Guaranteed	22.7

Q6: Preferred Local Network type

Baseband/CSMA-CD	44.8
Broadband/CSMA-CD	29.5
Baseband/Token	15.4
Broadband/Token	10.3

Q7: Traffic Types

Data and Graphics	45.7
Data only	25.1
Data/Graph/Video	17.2
All four	12.0

Q8: LN Compatibility Requirements

Multiple Vendor	69.1
Single Vendor	30.9

Q9: Gateway requirements

Dissimilar Networks	46.2
Similar Networks	34.9
Both	18.9

Q10: LN Standard requirements

Yes	93.5
No	6.5

Whilst this survey is not definitive it does provide very useful information on the requirements and interests of local network users. In particular standardisation and multiple vendors appear very important issues. It would also appear that many users are not particularly worried about the baseband v broadband or CSMA/CD v Token Passing type of debate. More information is given in the December issue of the Localnetter Newsletter.

13. Conclusions

Local networks cover a wide variety of systems and we have concentrated upon those which have reasonably high speeds at the present time. The situation in the U.K. over the next few years seems likely to be heavily influenced by the recent Ethernet standard discussed earlier and Cambridge Rings and other Ring based systems such as those from Xionics and perhaps Racal. It will be interesting to assess the Universe project when this is completed but early reports appear to be most encouraging.

One of the most important factors behind these developments is the emergence of LSI components for most of the local network hardware. This will bring the benefits of stability, reliability and low costs to users of these systems particularly for baseband systems. Ethernet CSMA techniques however are applicable on Broadband based systems though here the advantages of LSI may take longer to appear. Additionally there are speed restrictions on Baseband Ethernets which do not apply to rings, and the latter also have a distinct advantage in relation to the straightforward use of fibre optics.

If this mixed Ethernet/Ring scenario proves valid, then it will be important to have gateways between these networks. The SERC initiative at the University of Strathclyde is important in this context. As further developments in rings are carried out in the U.K. thus it will be crucial to put them into silicon at the appropriate time, test them out and tackle the international standards scene in a determined manner; speeds in the 50/100 megabit range appear quite feasible.

Another matter which may become important is that of encryption particularly for security purposes and the work at the National Physical Laboratory and in the Universe Project is pertinent in this regard.

In a more speculative vein it will be interesting to see if IBM decide to market a local area network product. As stated earlier they have published material on an experimental token ring [4] and this has been submitted to the IEEE802 committee. A possible complication is the patent position on token rings which may affect the attitude of the standards bodies.

A comment upon recent developments in Japan is perhaps in order. At least three companies, including Fujitsu and NEC, have announced local area network products involving fibre optic token ring, at speeds in the range 18 to 30 megabits per second, some of these can cover large distances, e.g. up to 100 kms, which is far in excess of the 3/4 kms diameter local networks given in our definition earlier in this paper.

In conclusion it appears that Local Networks are being increasingly used, we should have some stability once the LSI versions are available in 1983, though developments will certainly continue. Finally some standards de-facto are beginning to emerge.

14. REFERENCES

Open Literature

1 Flint, D., The Data Ring Main, John Wiley, 1983 (to appear)

2 Local Area Networks (Report to Focus Committee), Department of Industry, London, 1982

3 West, A., and Janson, P. (eds), Local Networks for computer communications, North Holland, 1981

4 Ravasio, P.C., Hopkins, G. and Naffah. N., Local Computer Networks, North Holland, 1982

5 Needham, R.M., and Herbert, A.J., The Cambridge Distributed Computing System, Addison-Wesley, 1982

6 Spratt, E.B., Local Area Networks: Management and Quasi-Political Issues, Proc ICCC82 (see Williams, M.B.)

7 Tannenbaum, A., Computer Networks, Prentice-Hall, 1982

8 Williams, M.B. (ed), Pathways to the Information Society, Proceedings of the Sixth International Conference on Computer Communication, North Holland, 1982

9 Local Networks and Distributed Office Systems, On line Conferences, 1981

10 The Ethernet, A Local Area Network-Data Link Layer and Physical Layer Specification, 30/9/80 (Digital, Intel and Xerox).

11 Local Area Networks. Report to the Focus Committee. Department of Industry, August 1982.

12 Clark, D.D., Program K.T. and Reed, D.P., "An introduction to local area networks", Proc IEEE, 66, 11, Nov 1978.

13 Cambridge Ring 82 - Interface Specification

14 Cambridge Ring 82 - Protocol Specification, Nov 1982

15 Rainbow Protocols

Green Book	TS29 Terminal Protocol
Red Book	Job Transfer and Manipulation Protocol
Yellow Book	Transport Service
Blue Book	File Transfer Protocol
Gray Book	Electronic Mail

Note. Copies of the references 13, 14 and 15 may be obtained from:

Joint Network Team,
SERC Rutherford Appleton Laboratory,
Didcot
Oxfordshire

16 Metcalfe, R. and Boggs, D., "Ethernet: Distributed Packet Switching for Local Computer Networks", CACM 19, 1976.

17 Saltzer, C., Clark, D.D. and Pogran, K., "Why a Ring", Proc 7th Data Communications Symposium, Mexico City, Oct 1981.

18 ISO 7498 Data Processing - Open Systems Interconnection-Basic Reference Model - obtainable from:

British Standards Institution
2, Park Street
London W1A 2BS

Consultancy Reports

Local Area Networks,
Xephon Technology Transfer Ltd, 1981
Kings House,
King Street,
Maidenhead,
Berkshire SL6 1EF

Localnetter Designers Handbook, 1982
Localnetter Newsletter (published monthly)
- both available from:

Architecture Technology Corporation,
P.O. Box 24344
Minneapolis, Minnesota 55424,
U.S.A.

Yankee Group Report on Local Communications,
C/IS Communications/Information Systems,
Regal House,
Lower Road,
Chorley Wood,
Rickmansworth,
Herts.

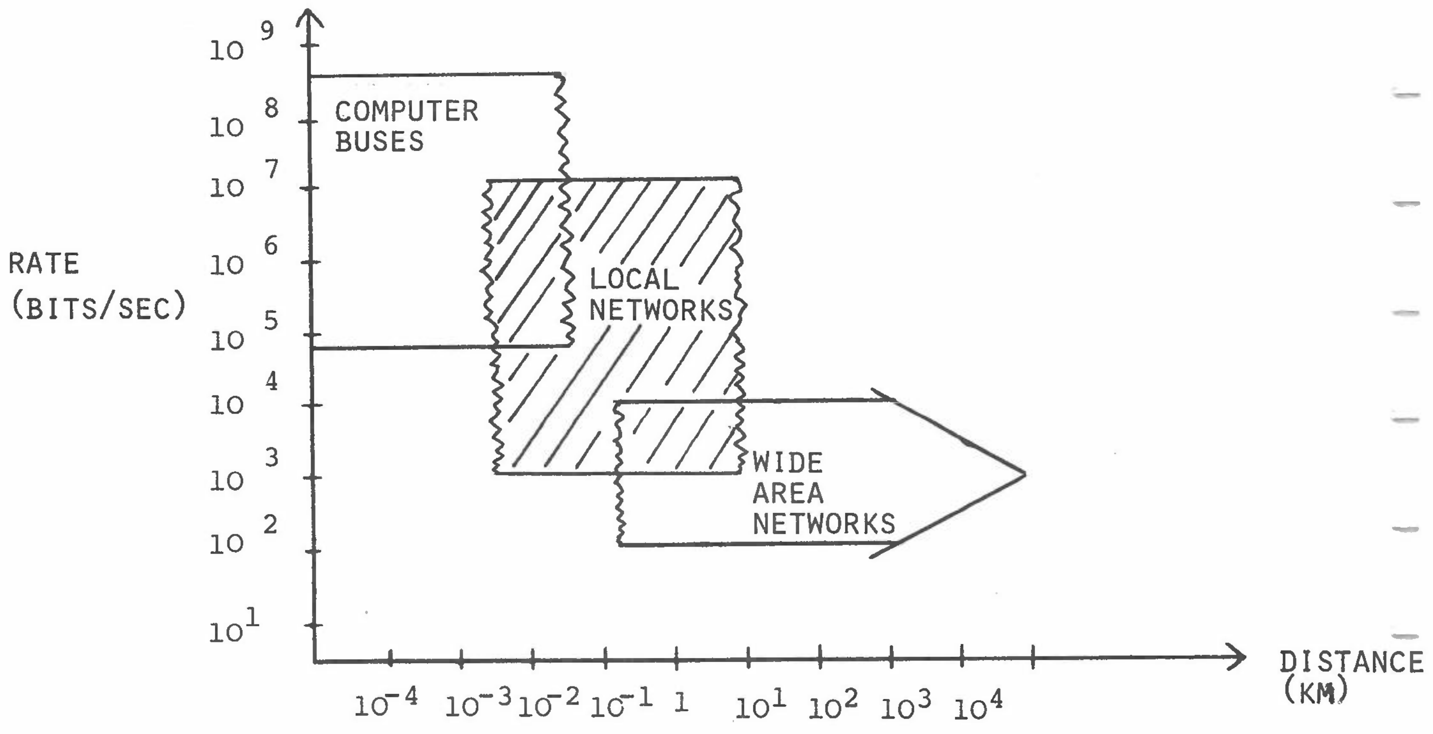


FIG. 1 COMPUTER NETWORKS

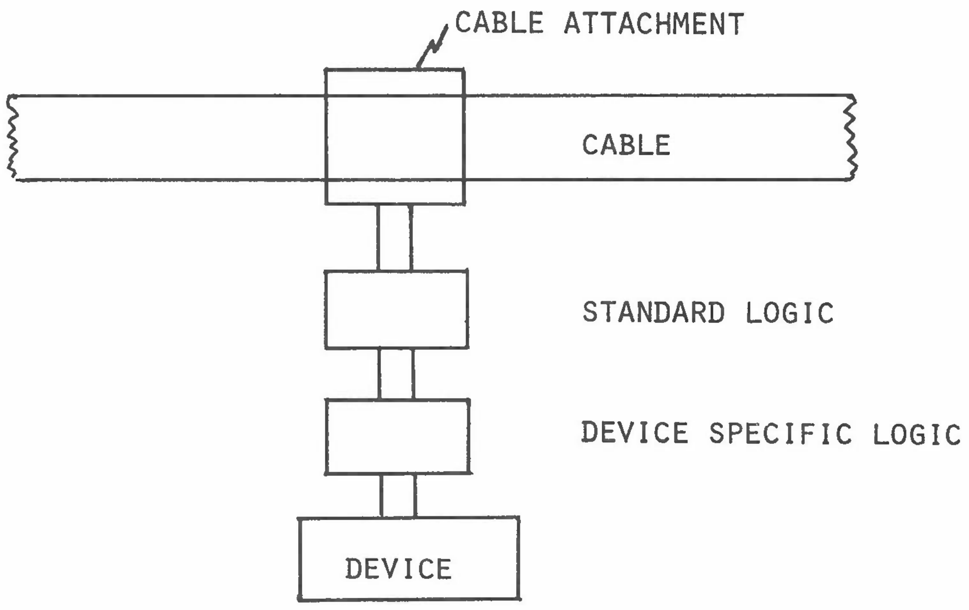


FIG. 2 TYPICAL LOCAL NETWORK NODE

FIG. 3 TOPOLOGIES

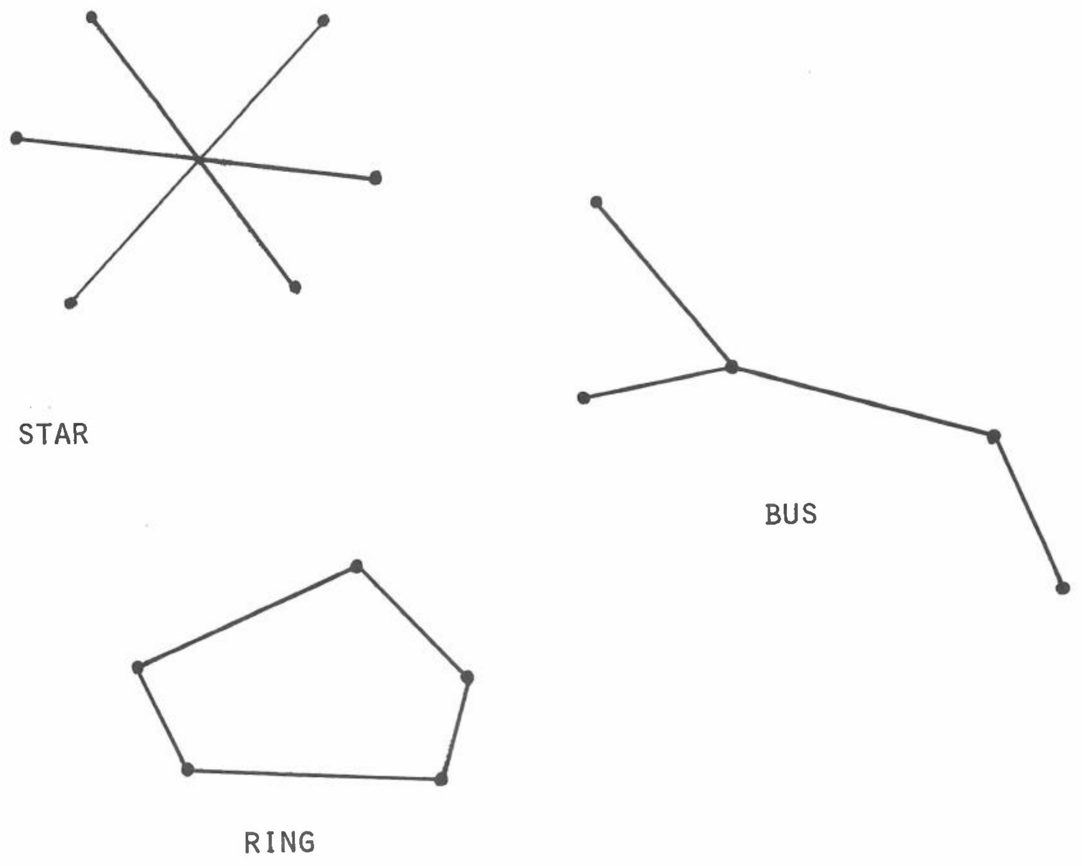


FIG 4 CAMBRIDGE RING NODE

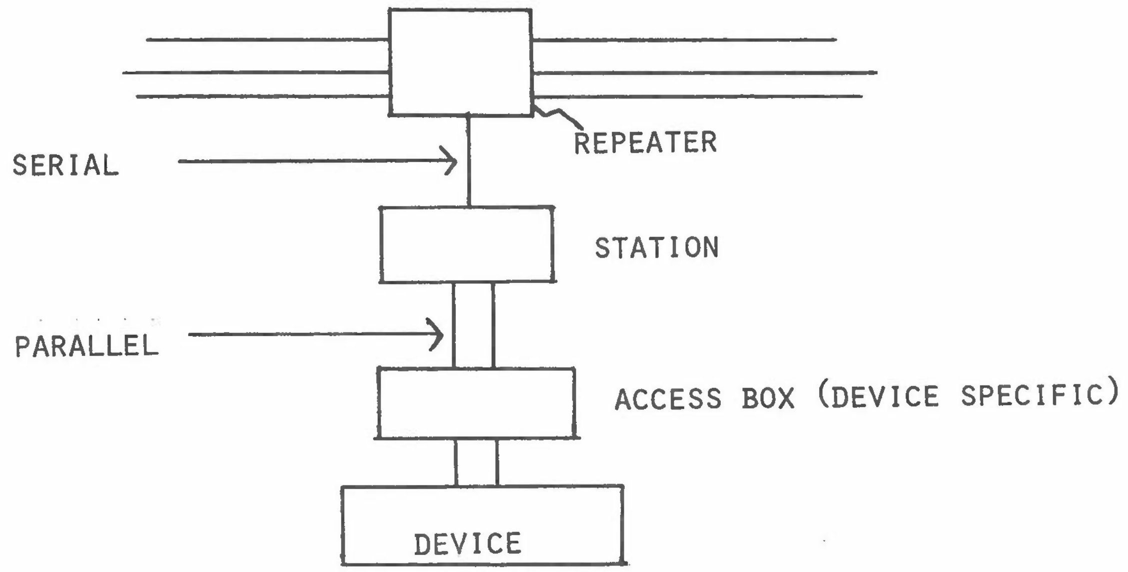
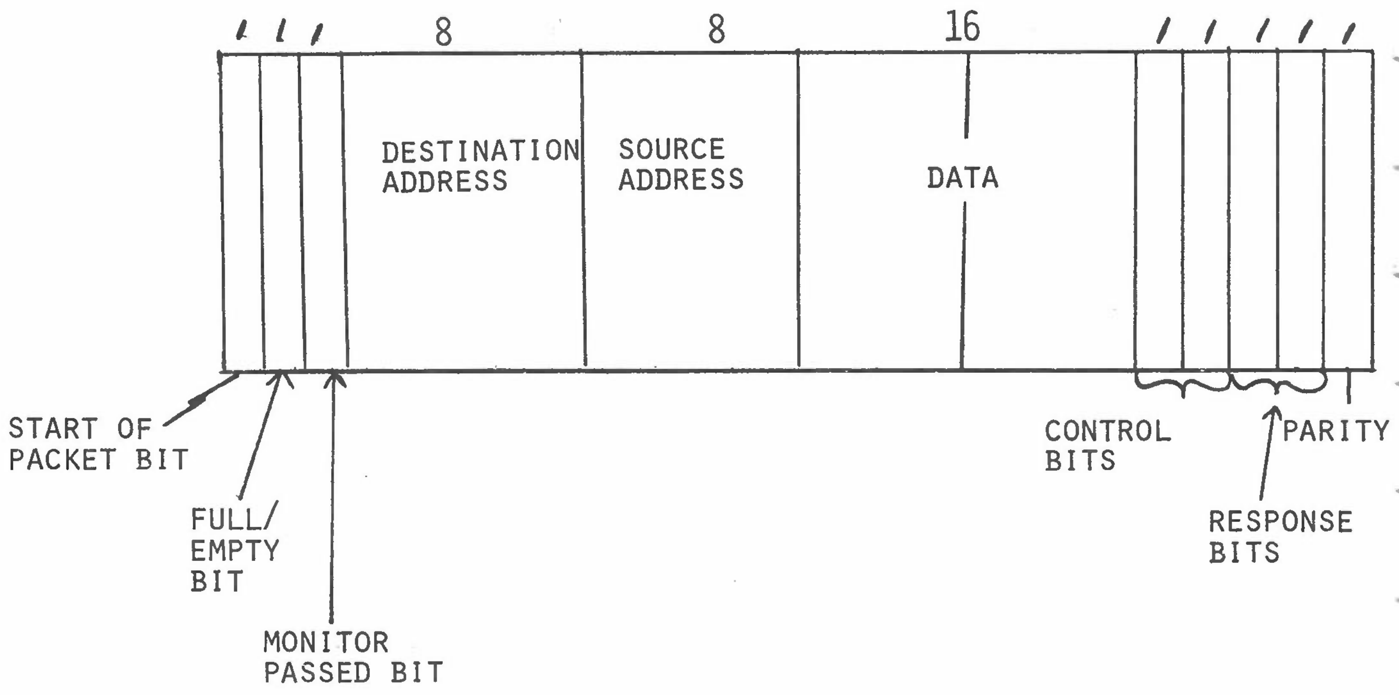
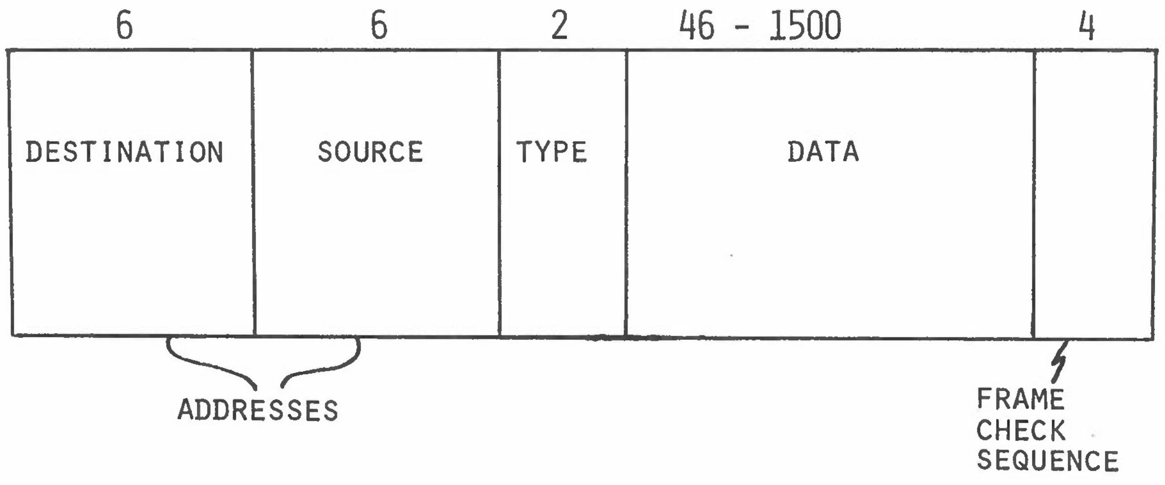


FIG. 5 CAMBRIDGE RING PACKET



UNITS OF BITS

FIG. 6 ETHERNET PACKET (DIX)



UNITS OF BYTES (OCTETS)

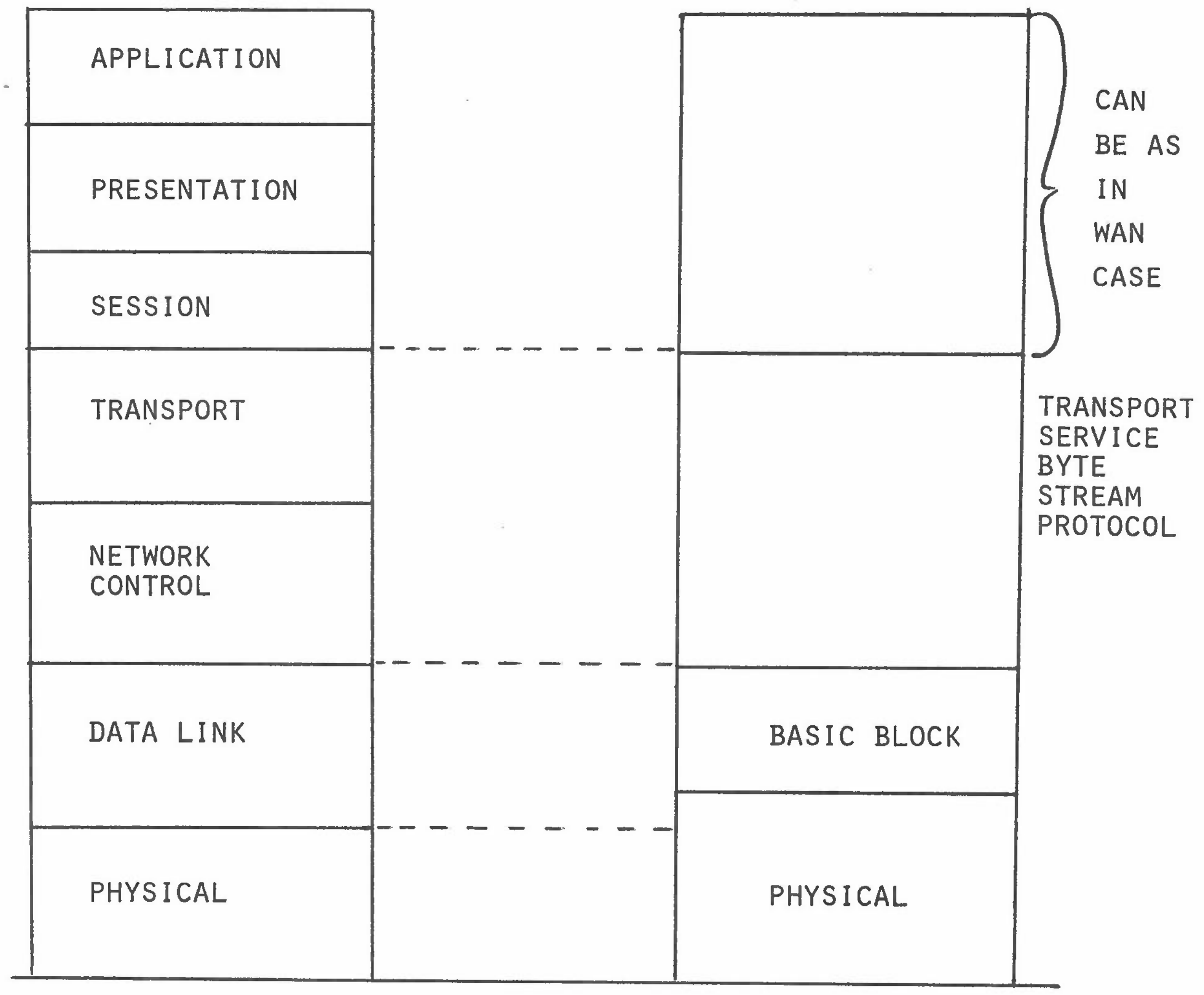


FIG. 7 ISO/OSI REFERENCE MODEL AND LOCAL NETWORKS - SPECIFIC EXAMPLE OF A CAMBRIDGE RING

SERC INDUSTRIAL DISTRIBUTED

COMPUTING SYSTEMS CONFERENCE

RINGS AND THINGS

BRIAN SPRATT
DIRECTOR, COMPUTER LABORATORY
UNIVERSITY OF KENT

- PERSONAL VIEWS

- EMPHASIS ON DEVELOPMENTS IN ACADEMIC COMMUNITY

    - UNIVERSITIES
    - POLYTECHNICS
    - SCIENCE AND ENGINEERING RESEARCH COUNCIL (SERC)

TWO TYPES OF COMPUTER NETWORK

- LOCAL
- WIDE AREA

LOCAL:	DIAMETER UP TO 2/3 KM
	TOTAL DATA RATE $>$ 1 MEGABIT PER SEC
	OWNED BY ONE ORGANISATION
WIDE AREA:	CAN BE INTERCONTINENTAL
	TYPICAL DATA RATE $\sim$9600 BITS PER SEC
	INVOLVE PTT'S (EG BRITISH TELECOM)

WHY ARE LOCAL NETWORKS IMPORTANT?

1) REQUIRED TO EXPLOIT ADVANTAGES OF FUNCTIONALLY DISTRIBUTED COMPUTING

COMPUTERS IN NETWORK DEDICATED TO SPECIFIC FUNCTIONS

- TERMINAL HANDLING
- DATA BASE MANAGEMENT
- STORAGE OF FILES
- PRINTING
- CONTROLLING PROCESS CONTROL EQUIPMENT

REASON FOR RELEVANCE TO SERC DCS PROGRAMME

2) USED TO INTERCONNECT

COMPUTERS
TERMINALS
PERIPHERALS (EG PRINTERS)
WORK STATIONS
REMOTE/LOCAL FACILITIES

CAN BE USED AS BASIS OF A COMPUTER SERVICE FOR TEACHING/
RESEARCH/ADMINISTRATION

JOINT NETWORK TEAM	SERC
	COMPUTER BOARD

DEVELOPMENT PROGRAMME

- STANDARDS

BANDWIDTH NOT SCARCE RESOURCE

SERC COMMON BASE POLICY

FORTRAN 77

PASCAL

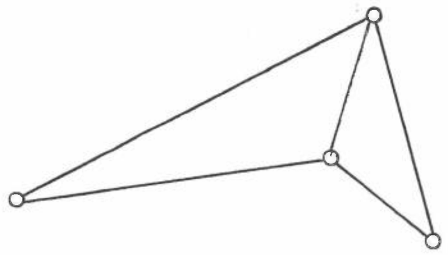
GEC 4000)
) MINICOMPUTERS
PRIME   )

ICL PERQ WORKSTATIONS

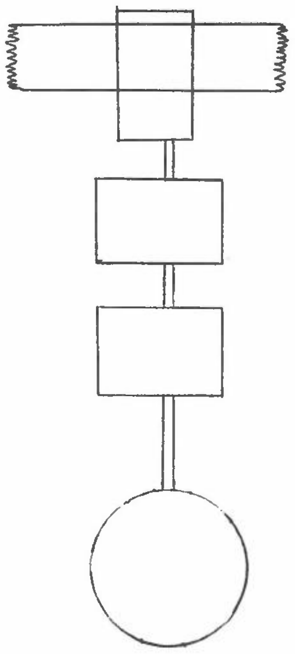
CONNECTED BY

CAMBRIDGE RING LOCAL NETWORKS

WHICH IN TURN ARE INTERCONNECTED USING WIDE AREA NETWORKS



4 NODE LOCAL NETWORK



TYPICAL NODE ON LOCAL NETWORK

PROTOCOL: PRECISE SET OF RULES ENABLING COMPUTERS TO COMMUNICATE EFFECTIVELY

TECHNOLOGIES FOR LOCAL NETWORKS

MEDIA - SERIAL TRANSMISSION

- CABLING
- BROADBAND AND BASEBAND
- PASSIVE
- NETWORK LAYOUT

SLOTTED RINGS

TOKEN PASSING

CARRIER SENSE MULTIPLE ACCESS (CSMA)

CABLING

TWISTED PAIR

COAXIAL

FIBRE OPTIC

- ENVIRONMENT
- BANDWIDTH
- NOISE IMMUNITY
- ENGINEERING/INSTALLATION ISSUES

BASEBAND

SINGLE SIGNALLING CHANNEL

ACCESS SIMPLE

SPEEDS 10-30 MEGABITS/SEC

100 MEGABITS / SEC UPWARDS FEASIBLE

BROADBAND

SEVERAL SIGNALLING CHANNELS

ACCESS COMPLEX - FREQUENCY AGILE MODEMS

CHANNEL SWITCHERS

CHANNEL SPEEDS 10 MEGABITS/SEC

CABLE TELEVISION TECHNOLOGY (CATV)

PASSIVE OR ACTIVE

NETWORK LAYOUT

NETWORK TOPOGRAPHY	WIRING
NETWORK TOPOLOGY	LOGICAL LINKS

MANAGEMENT

RELIABILITY

MAINTENANCE

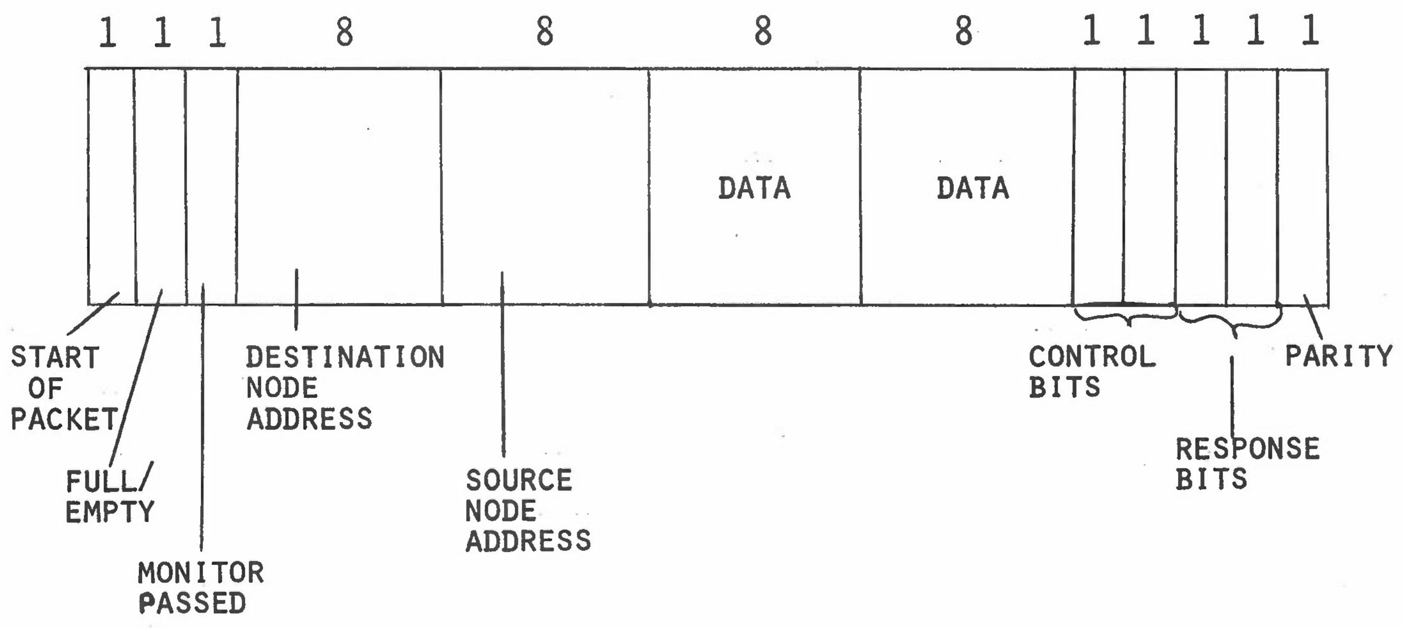
SLOTTED (CAMBRIDGE) RING

TWISTED PAIR

FIBRE OPTIC

REPEATER - STATION NODES

MINIPACKET - CR82



MONITOR NODE

NO HOGGING

NO SWAMPING

TOKEN PASSING

- HAVE TO POSSESS TOKEN TO TRANSMIT
- LOGICAL RING
- RINGS
  - VARIABLE LENGTH PACKETS
  - GOOD LINE UTILISATION
- BUSES (IEEE80Z)

CARRIER SENSE MULTIPLE ACCESS (CSMA)

BROADCAST

COAXIAL CABLE

COLLISION DETECTION

PROBABILISTIC

ETHERNET (CF ALOHA)

COMPARATIVE ISSUES

- DIFFICULT
- POSSIBLE TRENDS

# LOCAL NETWORK FORMS ONLY SMALL (BUT VITAL) PART OF TOTAL SYSTEM COST

# AVAILABILITY OF LSI COMPONENTS WILL REDUCE LN COSTS

# BASEBAND V BROADBAND

BASEBAND SPEEDS WILL INCREASE
FIBRE OPTIC TECHNOLOGY
CATV - FREQUENCY AGILE MODEMS
BROADBAND - VOICE
BUT PABX DEVELOPMENTS

ENGINEERING / MAINTENANCE

# MONITORING / MANAGEMENT

# SPECIFY REQUIREMENTS CAREFULLY

PROTOCOLS FOR LOCAL AND WIDE AREA NETWORKS

PROTOCOL - USED TO PROVIDE FOR AN ORDERLY EXCHANGE OF INFORMATION BETWEEN COMPUTER PROCESSES

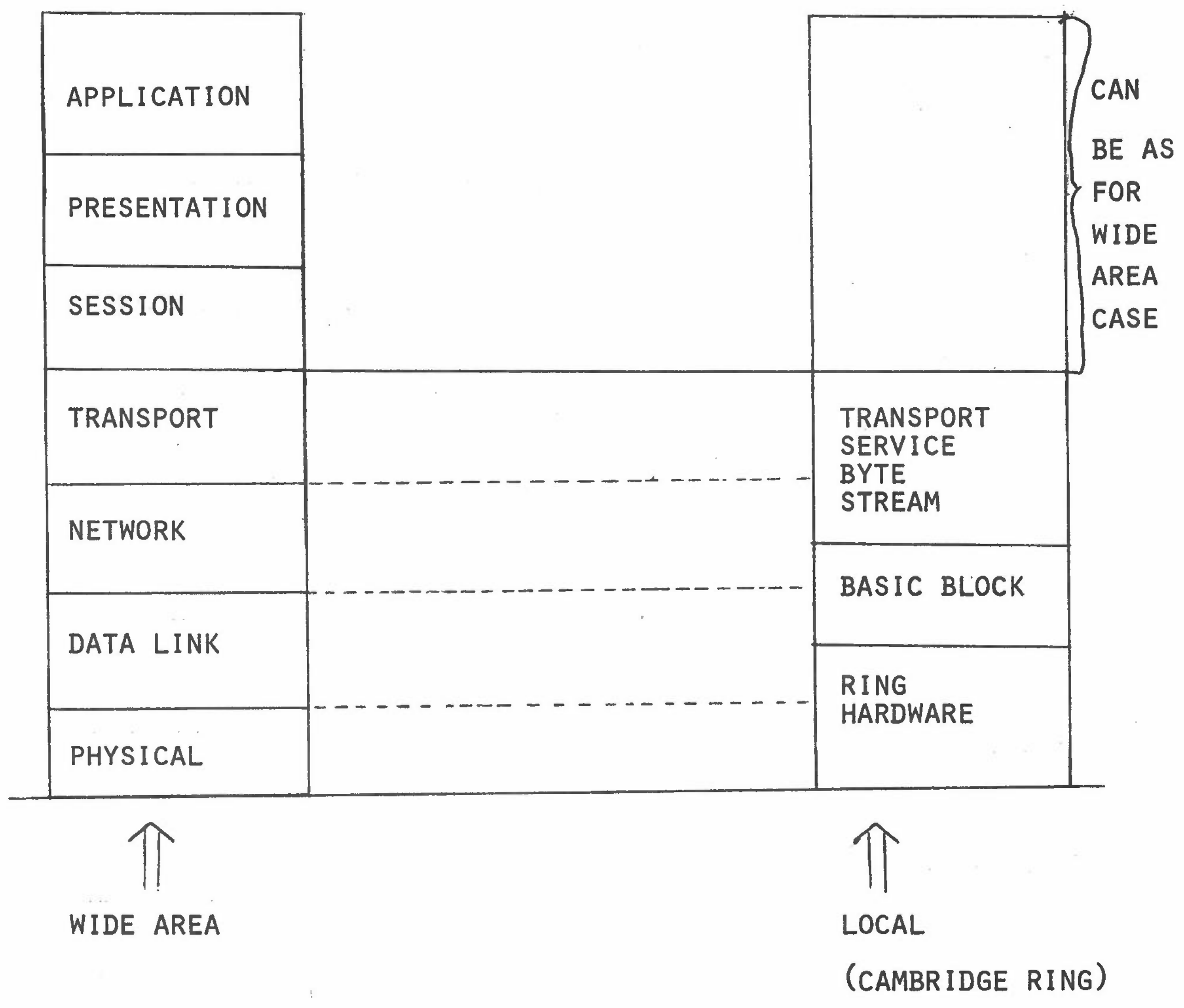
- STANDARDS

- REFERENCE MODEL FOR OPEN SYSTEM INTERCONNECTION (OSI)

  INTERNATIONAL STANDARDS ORGANISATION (ISO)

  7 LAYER MODEL

  DEVISED IN A WIDE AREA CONTEXT, BUT IDEAS VALID IN LOCAL NETWORKS



- BRITISH TELECOM SWITCHED STREAM 1 (PSS) SERVICE
  x25 (CCITT) LEVELS 1 TO 3

- CAMBRIDGE RING PROTOCOLS GIVEN AS EXAMPLE NOT EXACTLY AS FOR WIDE AREA CASE AT LOWER LEVELS

ISO/OSI REFERENCE MODEL, WIDE AND LOCAL AREA NETWORKS

ACADEMIC COMMUNITY PROTOCOLS

COLOURED BOOK PROTOCOLS (RAINBOW SERIES)

YELLOW BOOK	TRANSPORT SERVICE
BLUE BOOK	FILE TRANSFER
RED BOOK	JOB TRANSFER AND MANIPULATION
GREEN BOOK	TS29 TERMINAL PROTOCOL
GREY BOOK	ELECTRONIC MAIL
ORANGE BOOK	CAMBRIDGE RING PROTOCOLS

ETHERNET POSITION

ERROR RATES ON LOCAL NETWORKS

"STANDARDS" FOR LOCAL NETWORKS IN THE U.K.

DEVELOPMENTS:- CR82 CAMBRIDGE RING

IEEE802/DIX/ ETHERNETS

ECMA

CR82 AGREED RING COMPONENT SUPPLIERS ACADEMIC COMMUNITY

ACORN/ORBIS

LOGICA / VTS

SEEL

TOLTEC

LSI VERSION ~ LATE 1983

ETHERNET - XEROX

```
         - |D| EC   )
           |I| NTEL )   DIX
           |X| EROX )
```

IEEE802

ECMA - EUROPEAN COMPUTER MANUFACTURERS ASSOCIATION

ICL, CTL (OTL, NTL)

ACORN - ECONET

REAL TIME SYSTEMS LTD - CLEARWAY

INEXPENSIVE, LOW SPEED - STANDARDS?

EXAMPLES OF DEVELOPMENTS

CURRENT WORK IN U.K. - SERC

RINGS	● CAMBRIDGE - HIGH SPEED SLOTTED RINGS
	● UNIVERSE PROJECT
	CAMBRIDGE
	UNIVERSITY COLLEGE, LONDON
	LOUGHBOROUGH
	BRITISH TELECOM
	GEC MARCONI
	INTERCONNECTION OF CAMBRIDGE RINGS USING SATELLITE PLUS TERRESTRIAL LINKS
	● DISTRIBUTED COMPUTER SYSTEMS RESEARCH
	CAMBRIDGE
	KEELE
	KENT
	NEWCASTLE
	OXFORD
	QMC, LONDON
	STRATHCLYDE
	UMIST
	UCL, LONDON
	YORK

RINGS AND ETHERNETS	COMPARATIVE STUDY + GATEWAY	STRATHCLYDE
BROADBAND NETWORK	SUSSEX	

JOINT NETWORK TEAM CONTRACTS

TERMINAL CONCENTRATORS RING-RING BRIDGES RECONFIGURATION MONITORING	KENT
FIBRE OPTIC LINKS PRINTER SERVERS RING INTERFACES FOR VAX/UNIX VAX/VMS GEC 4000	OXFORD
RING COMPONENTS	EDINBURGH REGIONAL COMPUTER CENTRE

LOCAL NETWORKS AVAILABLE IN U.K.

CAMBRIDGE RINGS	LOGICA - VTS
	FIBRE OPTIC PRODUCT
	FILESERVER
	SEEL
	CABLE DUPLICATION
	ORBIS
	TOLTEC
"ECMA ETHERNET"	DIX/IEEE802
	ICL) LARGELY IMPORTED
	CTL (NTL, OTL))
	BASEBAND + BROADBAND
XIONICS RING	XIBUS - CLOSE COUPLED
	XINET - LOCAL NETWORK
	INTERFACES TO PSS
	.... MAINFRAMES (IBM, ICL, DEC)
RACAL - PLANET	

LIMITED NUMBER OF INSTALLATIONS 1982

MANY MORE 1983

CURRENT APPLICATIONS

- ACADEMIC COMMUNITY
  - RESEARCH VEHICLES FOR WORK ON DISTRIBUTED COMPUTING
  - BASIS OF COMPUTER SERVICES

    CAMBRIDGE

    KENT

  RINGS APART FROM STRATHCLYDE

  SUSSEX

  EDINBURGH

- NON ACADEMIC COMMUNITY

  INSTALLATIONS BASED ON IMPORTED PRODUCTS - "ETHERNETS"
  - THAME AND CASE

    (UNGERMANN BASS - NET ONE)
  - CTS

    (SYTEK)
  - XEROX

  OFFICE AUTOMATION - GOVERNMENT SCHEMES

USER SURVEY

- STANDARDISATION
- MULTIPLE VENDORS
- BASEBAND V BROADBAND
- CSMA/CD V TOKEN PASSING

CONCLUSIONS

U.K. SITUATION HEAVILY INFLUENCED BY

- CR82
  IEEE802/DIX/ECMA 'ETHERNET"
  XIONICS
  RACAL
- LSI COMPONENTS
- GATEWAYS FOR INTERWORKING

# NEED FOR MORE U.K. ACTIVITY ON #
INTERNATIONAL STANDARDS

IBM	TOKEN RING
JAPAN	FUJITSU
	NEC

FIBRE OPTIC TOKEN RINGS

18 - 30 MEGABITS/SEC

UP TO 100 KMS

LOCAL NETWORKS INCREASINGLY USED

SOME STABILITY WITH LSI

STANDARDS EMERGING

EVALUATING LOCAL AREA NETWORKS

Doug Shepherd

Computer Science Department

University of Strathclyde

1. Introduction

The past two years have seen a large increase in the commercial availability of local area networks. These systems are usually based on one of the following three architectures:- slotted ring, CSMA collision detection bus, or token ring. Examples of each type are the Cambridge Ring, Standard Ethernet and MIT's token ring respectively. A number of papers have appeared describing these architectures [1,2,3] and the paper by Spratt[4] gives a good overview of the systems available. Most of the manufacturers make similar claims for the performance and versatility of their local area networks and it is the purpose of this paper to present a cross section of the work that has been carried out in evaluating the performance of various local area networks. It is not our intention to give an exhaustive review of the literature but rather to select one or two results that will give the reader some feeling for the relative merits of the various local area network architectures.

In the first part of the paper we look at modelling, both analytical and simulation, in the second part we present some actual performance measurements for the Ethernet and Cambridge Ring respectively, in the third part we discuss other factors that should be considered besides performance, and finally we draw appropriate conclusions.

2. Modelling

A large number of papers have appeared in the literature on modelling various local area network architectures [5]. The majority use analytical techniques but there are a few that use simulation. The three papers we will discuss in some detail are the ones by Werner Bux[6], Blair and Shepherd [7], and Almes and Lazowska[8]. The first paper compares analytically four different types of LAN architectures namely: token ring, slotted ring, random access bus (CSMA with collision detection), and ordered access bus (MLMA reservation scheme). The second uses simulation to compare the Standard Ethernet[9] with the Cambridge ring. The work described in the third is unusual in that it first of all derives an analytical model of an Ethernet-like system and then simulates the Experimental Ethernet system.

2.1 Analytical Comparison of Four Types of LAN Architecture

The performance criterion that Bux investigates is the delay-throughput characteristic of the system. Delay is measured as the mean transfer time of the packets which he defines as the time interval from the generation of a packet at the source station until its reception at the destination. This means the transfer time includes the queueing and access delay at the sender, the transmission time of the packet, and the propagation delay.

Modelling of the Networks

In order to allow for direct comparison of the results, consistent assumptions are made with respect to traffic properties for all models. These are: packets are generated at the S stations according to a Poisson process and the packet lengths can be generally distributed. A header is added to every packet which contains control and addressing information. Bux draws the following conclusions from his study. The token ring performs almost ideally over the whole range provided the delay in each station is kept to a minimum. The slotted ring shows comparatively high transfer delay values due to the short slots of this type of ring, which mean that there is a high overhead for addressing and control information and the time needed to pass empty slots around the ring to ensure fair use of the bandwidth. The bus with CSMA and collision detection behaves ideally as long as the ratio of propagation delay to mean packet transmission time is low. If this ratio exceeds 5% the increase in collision frequency causes significant performance degradation. The MLMA ordered access bus shows slightly higher transfer delay than the token ring. This difference, which in most cases is insignificantly small, is caused by the overhead required for scheduling of the packets.

2.2 Simulation

Blair and Shepherd have carried out a number of simulation studies of the Cambridge Ring and Ethernet-like systems[7,10]. The one we will discuss here compares the Cambridge Ring with the DEC, Intel, Xerox Ethernet[9]. The Standard DIX Ethernet specification was used namely:-

Data Rate	10 Mbs	Slot time 512 bits
Jam Signal	32-48 bits	Interframe spacing 9.6-10.6 microsecs
Preamble	64 bits	Packet size 64-1518 bytes

The workload model consists of S stations. A Poisson arrival of messages is assumed with mean inter-arrival times t1 through tS. Constant message lengths L1 through LS are also assumed. A block consists of a header, route, data and checksum. A block is transmitted as one Ethernet frame or N ring minipackets. The error rate for each system is considered to be 1 in 10^7. Message destinations are random.

The following message statistics are collected:

1. Start time - arrival time in station queues
2. Select time - time message is selected for transmission
3. Finish time - acknowledgment successfully received.

From these the mean queuing times, service times and delays can be calculated.

The number of stations, propagation delay between stations and frequency of the systems can be varied. In the case of the Cambridge Ring the number of minipackets and the number of data bytes per minipacket can be set. For the Ethernet the inter-frame spacing, length of preamble and length of jam signal can be varied.

The frequency of both the ring and ethernet were set at 10 MHz with a 6 bit delay between stations. The number of minipackets in the Cambridge Ring model is set to be optimal. The decision rule is to minimize the minipackets without introducing wasted bandwidth[11].

A number of experiments were carried out by varying the number of stations in the system while keeping the message length constant at 16 bytes. The results obtained show that as the number of stations increase the Ethernet is only better than the Cambridge Ring at low loads. For 32 stations the Cambridge Ring performs consistently better. The reason for this is that the collision window for the Ethernet is longer for larger networks increasing the chance of collisions whereas the addition of extra stations to the Cambridge Ring can be compensated by the extra packets that can be accommodated.

The simulators were then run using different message lengths. As the message lengths increase the performance of the Ethernet system is superior to the Cambridge Ring and in addition is more stable i.e. expected delay does not degrade sharply as load increases.

Satisfactory delay characteristics in LANs depend not only on low, stable expected delay but also on low variance of delay times. The variance of delay in the two systems was compared by plotting the ratio of standard deviation to mean for delay times against total offered load. Ethernet clearly has higher variance at all level of loads. There are two reasons for this :

1. It has been shown [12] that the backoff algorithm achieves stability for Ethernet at high loads at the expense of a kind of last come first served scheduling[8] resulting in high variance of delay times.
2. The Cambridge Ring has guaranteed maximum and minimum transmission times. The low level protocol implies that a station at worst gets 1/n(m+2) and at best gets 1/(m+2) in an n station, m minipacket ring.

The paper concludes that for most configurations Ethernet has a lower expected delay than the Cambridge Ring. The major reason for this is the overhead of the minipacket protocol in the Cambridge Ring. However the Cambridge Ring has the desirable property that delay characteristics do not depend on the size of the network or on the message lengths.

Almes and Lazowska use a simple analytic model based on 1/q control to study Ethernet-like systems, where q is a measure of the instantaneous load on the communication medium. They also simulate the Experimental Ethernet and from the results of their simulation conclude that their analytical model is acceptable. They also draw the following conclusions:-

Ethernet and other networks based on the 1/q model are stable.

The Ethernet has considerable variance in response time. This variance does not, however, make it unsuitable for "soft real-time" applications at moderate average loads.

The performance of the systems is quite sensitive to packet size distribution. Higher technologies e.g optical fibres, will provide greatest benefit for applications that can use large packet lengths.

They also conclude that there might be some benefit in using a back-off algorithm based on an estimate of q from information available to the stations.

3.Some Practical Measurements

Very few papers have appeared containing actual performance measurements of existing systems. An exception is the paper by Shoch[12] which considers the Ethernet at Xerox Parc and a paper by Temple[13] which examines the Cambridge Ring system at the University of Cambridge.

3.1 Traffic Measurements of Ethernet

At Xerox Parc there are a number of interconnected Ethernet systems which have been providing a service for several years. They use a coaxial cable running at 2.94 MHz. The particular local Ethernetwork chosen for the measurements spans about 1800 feet and connects over 120 machines. These machines include a large number of single-user stand-alone computers, two time-sharing servers, numerous shared printers and fileservers as well as several gateways. Applications include: file transmission to the printers, access to shared data-base systems and terminal access to time-sharing machines.

To conduct the measurements a series of specialised test and monitoring programs has been constructed to assess the behaviour of the network. There is a promiscuous station which can receive all of the packets passing by. The measurements were taken using this passive technique.

One of the first results obtained was that one damaged packet in about 6000 was detected, and this resulted in the design of a new interface. Using the new interface a packet error rate of 1 in 2,000,000 packets was achieved.

Performance Under Normal Traffic Loads

The utilisation of the system over a full 24-hour period ranges from 0.60% to 0.84%. During the busiest hour this rises to 3.6%, busiest minute 17% and busiest second 37% This verifies the design assumption that computer applications tend to produce a bursty pattern of requests.

Most of the packets sent through the system are short ones, but most of the total volume is carried in the large packets.

Performance Under High Load Conditions

The previous section discussed the system under normal operating conditions. Further growth of new systems will increase the load on the net and the system ought to be able to handle short term bursts at very high load.

To enable this to be tested a set of test programs was constructed to generate artificially high levels of traffic. Using a special control program these are loaded into idle machines on the net and then used to produce a specified offered load to the network. As the total offered load increases from 0% to 90% channel utilisation matches it perfectly: all the traffic gets out correctly and under high loads the Ethernet system remains stable.

From these results it can be concluded that:

1. The error rates are very low, and few packets lost.
2. Under normal load, there are very few collisions.
3. Under heavy load there are more collisions but the collision mechanisms work well and channel utilisation remains high.
4. Even under heavy loads the Ethernet channel does not become unstable.

The last result would suggest that the several proposals for complex control schemes would offer little benefit for the increased complexity.

3.2 Traffic Measurements of the Cambridge Ring

Some measurement work has been done at the University of Cambridge Computer Laboratory by Temple. The system consists of a Cambridge Ring operating at 9.8 Mbit/sec, with 3 minipacket slots and a gap of 3 bits. The ring is used to support the Cambridge Distributed Computing System[14] and also has a number of machines connected to it, such as the IBM 370, which provide a service for the general user. One of the devices on the ring is the fileserver[15] which as well as providing filing capabilities for the processor bank is used as a paging device for the CAP machine. i.e. swapping is carried out across the Ring.

A special device called a Traffic Monitor has been built which can be connected to the ring via a repeater. The repeater can either be a stand-alone one or one connected to a station. The traffic monitor can be attached to a standard vdu which allows particular patterns to be entered for matching against minipackets. The software also allows histograms of traffic to be displayed on the vdu. A complete description of the monitor can be found in the report by Balfour[16].

The results from a typical experiment to see how many packets were accepted were that 86% are accepted, 2% unselected, 5% ignored, and 7% busied. The ring utilisation was 3.4%.

Most of the time the ring utilisation ia about 1% although bursts of up to about 20% were observed when a single station transmits at ring speed. For a medium term transaction, loading a processor bank machine, the utilisation is of the order of 10%. Over a longer period (30 mins) the typical utilisation is 3% and over a 24 hour period is 1.3%. The CAP is the only machine working at ring speed and one can see the bursts of CAP activity over a 0.1 sec time period. Over a 10 sec time period bursts are eliminated and utilisation clusters around a mean of 2.7%.

4.Other Considerations

Although performance is an important factor in considering a local area network one thing that is perfectly clear from the previous sections is that all the systems discussed have more than ample bandwidth for currrent applications. In fact it is the belief of this author that in the case of Distributed Operating Systems the type LAN architecture you choose is irrelevant. In fact at Strathclyde we have developed a network operating system, called MIMAS, which runs on top of both an Ethernet-like system and a Cambridge Ring[17]. Therefore it should be other factors that influence the choice of LAN used. We discuss a few of these factors below.

Internetworking

It seems extremely likely that any large organisation will have a number of separate installations consisting of devices connected together by their own local network. In order to gain the full potential of networking and to maximise the sharing of expensive facilities such as disc storage it will be necessary to link these networks together. In addition it may be desirable to link the local area network to a wide area network, typically through the X25 access protocol.

Because of the variable size of their basic packets the token ring and the Ethernet have a clear advantage over the slotted rings with their smaller fixed length packets. For instance, in the case of the Cambridge Ring which has only 8 bits for the destination address the limit on the number of stations it can address is 256. This means that it has to use a local addressing scheme. Therefore, any gateway linking Cambridge Rings together must be able to transform from a local address to one recognised by the rest of the networks. The token ring and Ethernet, however, can have a larger destination and source field. In their case a global addressing scheme could be adopted using a standard 48 bit address field which would allow every station to have a unique address over all the networks. This has been proposed in the paper by Dalas and Printis[18], who call it a universal address. The use of universal station numbers in an internetwork provide for reliable and manageable operations as the system grows, as machines move, and as the overall topology changes.

Flow Control and Guaranteed Response.

The token ring and Ethernet have no low level flow control. A packet is sent to a station and the sender has no idea whether it has been accepted or not. Any acknowledgment must be provided by the higher level protocols. Experience is showing that it is the high level protocols which slow the systems down. The Cambridge Ring, however, indicates in the reponse bits whether or not the minipacket has been accepted. If for instance the response bits indicate that the destination was busy the minipacket can be retransmitted.

With an Ethernet system there is no guaranteed response time. In the case of a token ring the maximum response time will be function of message lengths as well as the number of stations. The slotted ring, however, will have maximum response time which only depends on the number of stations in the system. This might be important in some real-time situations, for instance if the local area network is used to link some form of control system together.

Another example is the use of mixed voice and data traffic over the network. If the network is very lightly loaded then an Ethernet could handle voice as well as data[19] taking advantage of the fact that there is considerable redundancy in the digitised speech. However, if the load increased by having a large number of calls going on at the same time then the information lost would be unacceptable. The Cambridge Ring, however, would have the same characteristics whatever the load[20].

Cost

An important factor in evaluating any LAN will be its cost. In the opinion of the author there are a large number of applications of LANs that do not require a 10 Mbyte/sec bandwidth. Xerox Parc, for instance, have been operating with a 3 Mbyte/sec system for a number of years and the measurements carried out show that there is considerable spare capacity. This would appear to indicate that systems built using cheap off-the-shelf components operating at 1-2 MBits/sec would be quite suitable for a large number of cases, for example a shop floor reporting system[21].

High Level Protocols

In this paper we have concentrated on the low level characteristics of the LANs and no attempt has been made to discuss the type of high level protocols they support. It is the design and implementation of suitable protocols for systems that is now the major problem in LANs. It is not clear that the underlining architecture has much of an influence on the higher level protocols and this is one of the main current areas of research. There is no doubt that these are the critical factors in LAN performance.

5.Conclusions

In this paper we have presented some results from studies in evaluating the performance of LANs. The results obtained from the analytical and simulation studies give some indication of the likely performance of the various systems. However, they should be treated with caution as the traffic profile used in them bears little relation to those found in practice. The simulation studies, on the whole, seem to produce results that are nearer to those found in practice. The actual measurement figures are extremely interesting and seem to suggest that the predicted collapse of the Ethernet does not occur and that back-off is no problem. On pure performance grounds the token ring would appear to be best but other factors besides performance should be taken into consideration, for instance whether the system would support a real-time application.

Finally we would like to quote from an interesting paper by Saltzer and Clark[22]: "Considering the various technical arguments it appears that one cannot make a clear case for either the contention-controlled broadcast net or the ring technologies. Both approaches have good arguments in their favour, and it is likely that such issues as ease of installation, maintenance, and administration will dominate the technical issues".

References

[1] M.V.Wilkes,D.J.Wheeler,"The Cambridge Digital Communication Ring",Local Area Comms Network Symp, Mitre Corp and National Bureau of Standards,Boston,May 1979.

[2] R.M.Metcalfe,D.R Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks",CACM Vol 19,No 7,July 1976

[3] D.D.Clarke,K.T.Pogran,D.P.Reed, "An Introduction to Local Area Networks", Proc IEEE, Vol 66, No 11, Nov 1978.

[4] B. Spratt, "A Review of Available Local Area Networks", DCS Industrial Conference, Manchester, March 1983.

[5] H.A.Freeman and H.J.Thurber,"Updated Bibliography on Local Computer Networks", Comp Comm Review, Vol 10, No 3,July 1980.

[6] W.Bux, "Local Area Subnetworks : A Performance Comparison", Proc. of IIFIIP Working Group 6.4 International Workshop on Local Networks, Zurich, August 1980.

[7] G.S.Blair, W.D.Shepherd, "A Performance Comparison of Ethernet and the Cambridge Digital Communication Ring", Computer Networks, Vol 6, No 1, Feb 1982.

[8] G.J.Almes,E.D.Lazowska, "The Behaviour of Ethernet-like Computer Communications Networks ", Proc of 7th Symp on Operating Systems Principles, Dec 1979,pp61-81.

[9] The Ethernet,A Local Area Network,Data Link Layer and Physical Layer Specifications,Digital,Intel and Xerox, version 1.0, September 30,1980.

[10] G.S.Blair,D.Hutchison and W.D.Shepherd, "A Comparison of Ethernet-like and Ring Communication Systems ", Proc IEEE Vol 129, Pt E,No 4, July 1982.

[11] G.S.Blair, " A Performance Study of the Cambridge Ring ",Computer Networks,Vol 6,No 1, Feb 1982.

[12] J.F.Shoch,J.A.Hupp, " Measured Performance of an Ethernet Local Area

Network ",CACM,Vol 23, No 12, Dec 1980,pp711-721.

[13] S.Temple,"Performance Measurements on a Cambridge Ring", presented at DCS Cambridge Ring Modelling and Simulation SIG,Cambridge,March1981.

[14] R.M. Needham, A.J. Herbert, "The Cambridge Distributed Computing System", Addison-Wesley Publishing Company, 1982.

[15] J. Dion,"The Cambridge Fileserver", Operating Systems Review, 14, 26-35, 1980.

[16] J.Balfour, "A Performance Monitor for the Cambridge Ring", Internal Report,Department of Computer Science,Univ Strathclyde,July 1982

[17] G.S.Blair,D.Hutchison and W.D.Shepherd, " MIMAS- A Network Operating System for Strathnet", Proc 3rd Int Conf on Distributed Computer Systems, Florida,Oct 1982.

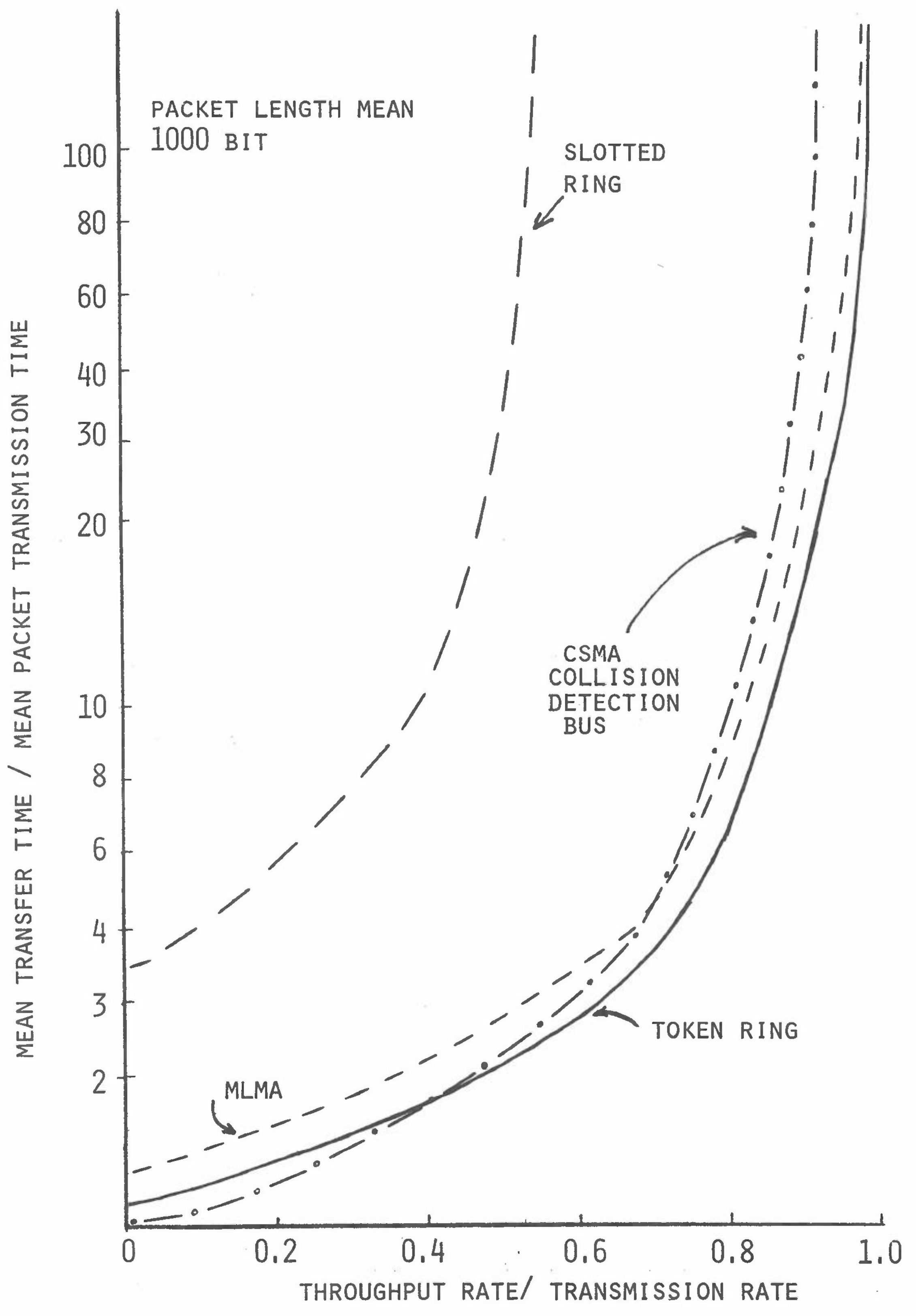
[18] Y.R Dalas and R.S Printis, "48 - Bit Absolute Internet and Ethernet Host Numbers", Seventh Data Communications Symposium,Mexico City, October 1981.

[19] J.F.Scoch, "Carrying Voice Traffic through a Local Network - a general overview ", presented at the IFIP WG 6.4 International Workshop on Local Area Computer Networks, Zurich, August 1980.

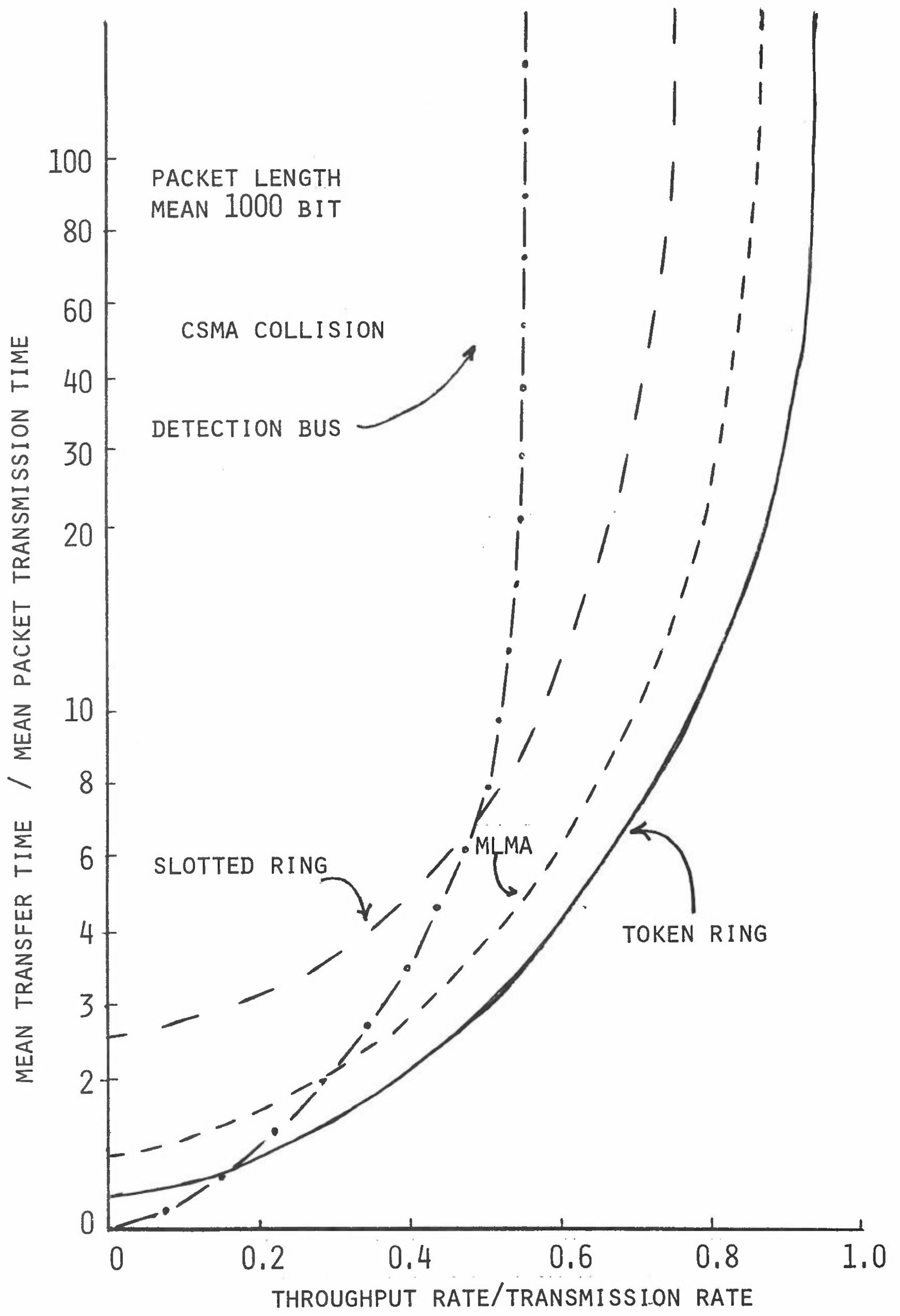
[20] I.M. Leslie, R. Banerjee, S.J. Love, "Organisation of Voice Communication on the Cambridge Ring",Online Conf. on Local Networks and Distributed Office Systems", London, May 1981.

[21] D. Hutchison, W.D. Shepherd, "Strathnet - A Local Area Network", Software and Microsystems, Vol 1, no 1, pp 21-27.

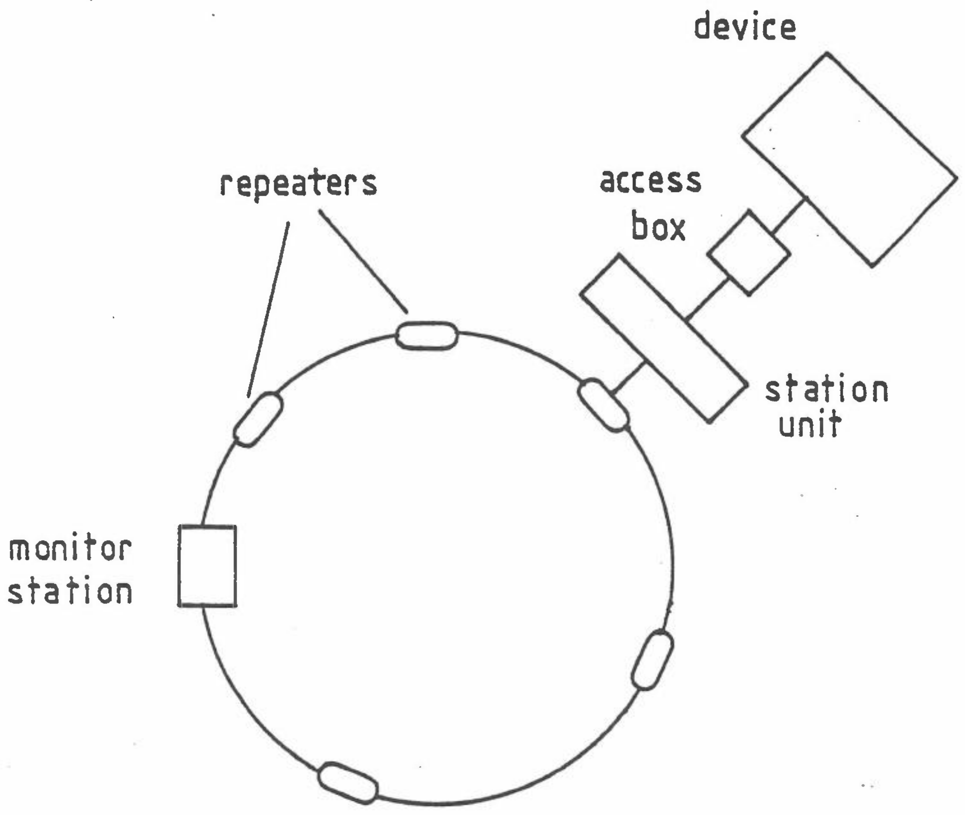
[22] J.H. Saltzer, D.D. Clark, "Why a Ring ?", Proc. Seventh Data CommunicationsSymposium, Mexico City, October 1981.



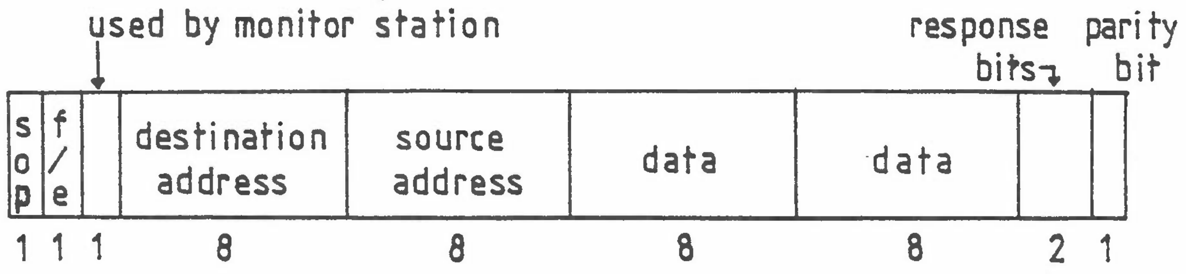
TRANSFER DELAY - THROUGHPUT CHARACTERISTICS OF THE FOUR LANS AT 1 M BIT/SEC



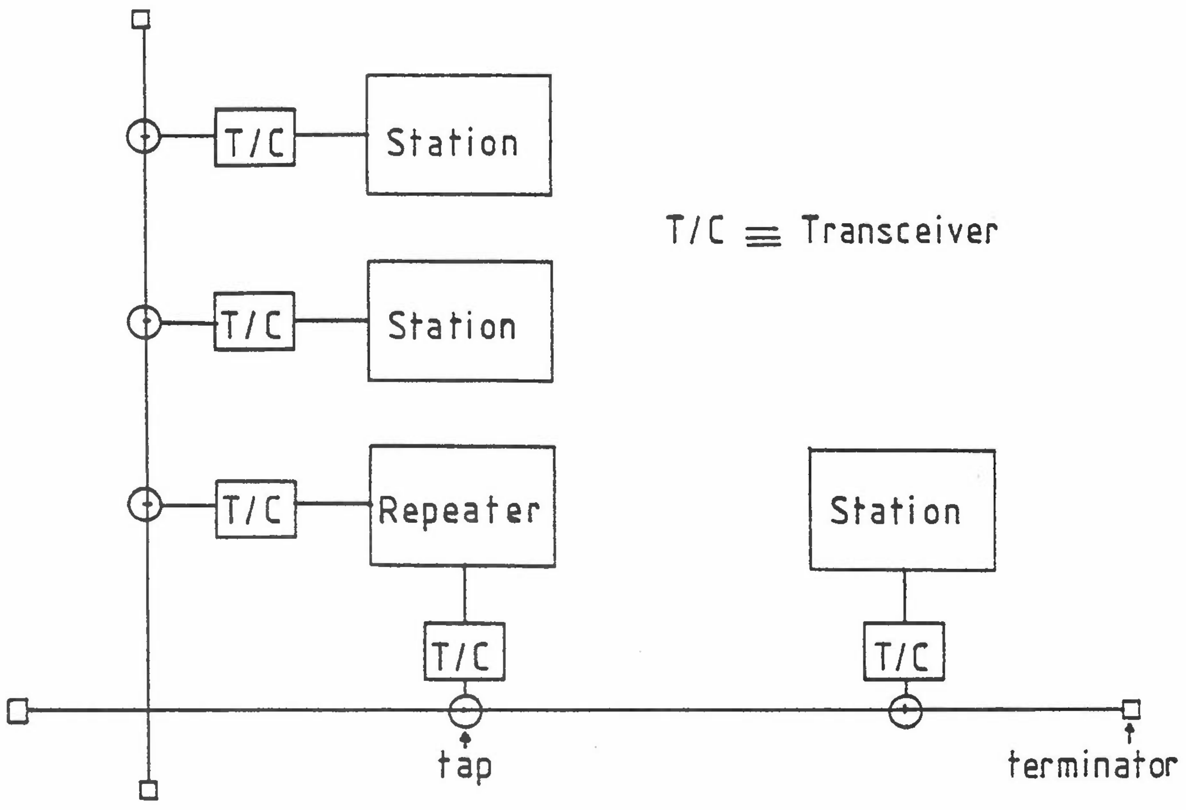
TRANSFER DELAY - THROUGHPUT CHARACTERISTICS OF THE FOUR SYSTEMS AT 10 M BIT/SEC



Ring Structure



Mini-packet Structure

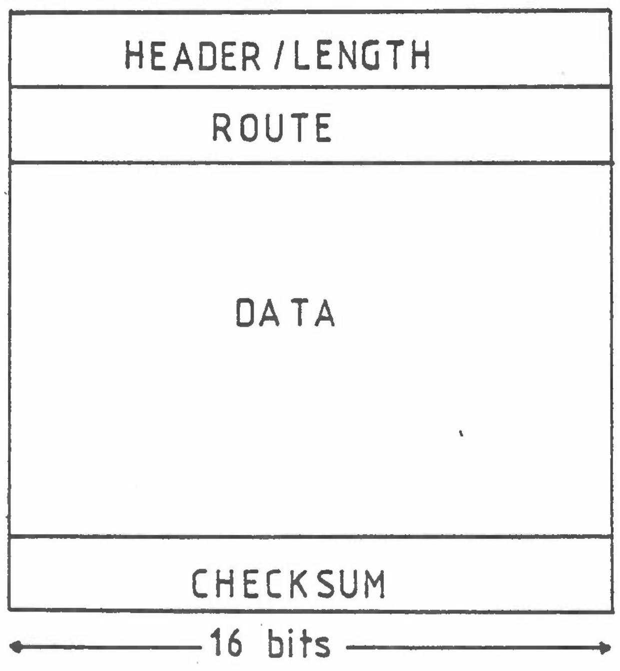


A Two Segment Ethernet

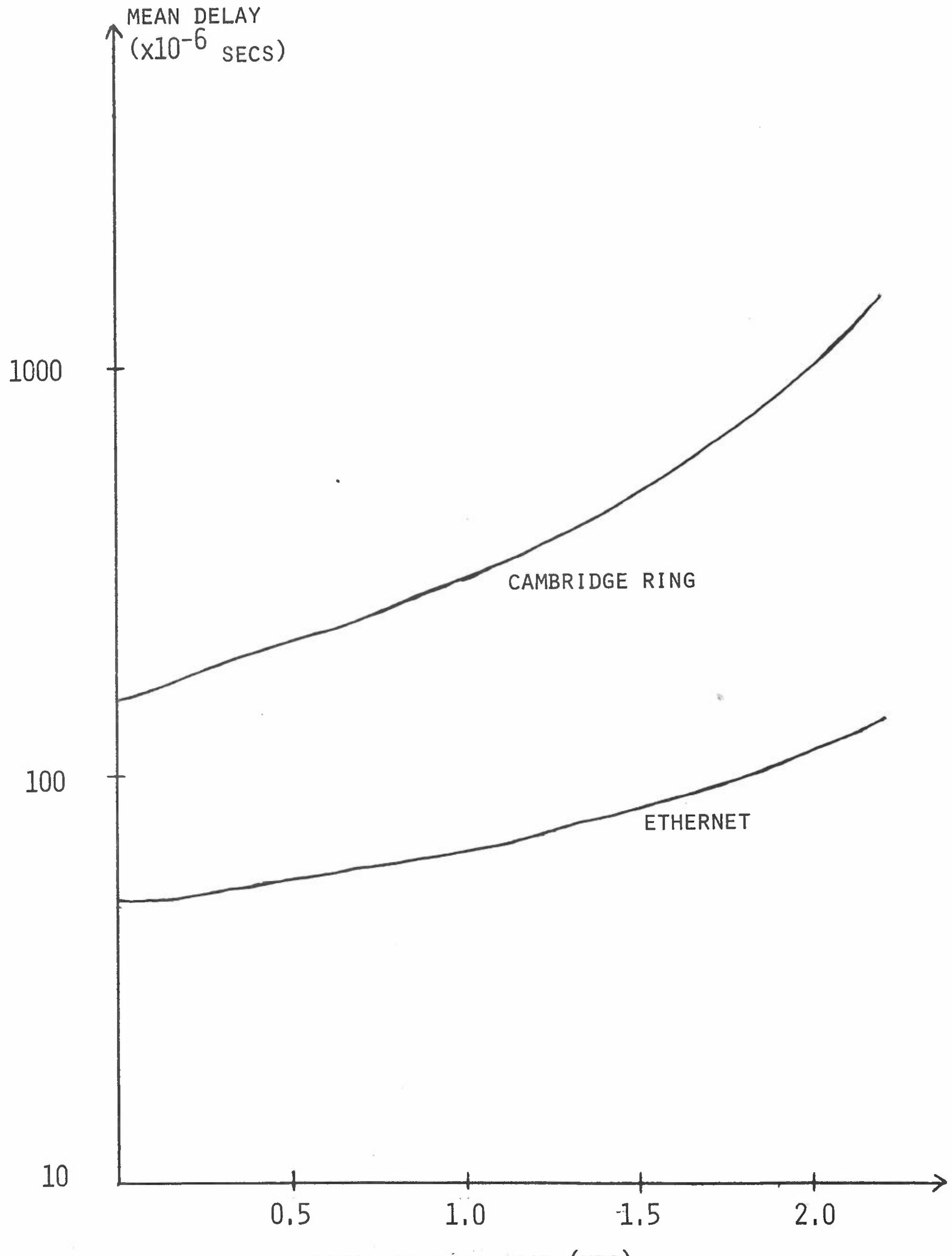
6 octets	6 octets	2octs.	46-1500 octets	4 octets
Destination	Source	Type	Data	FCS

octet = 8 bits FCS = Frame Check Sequence

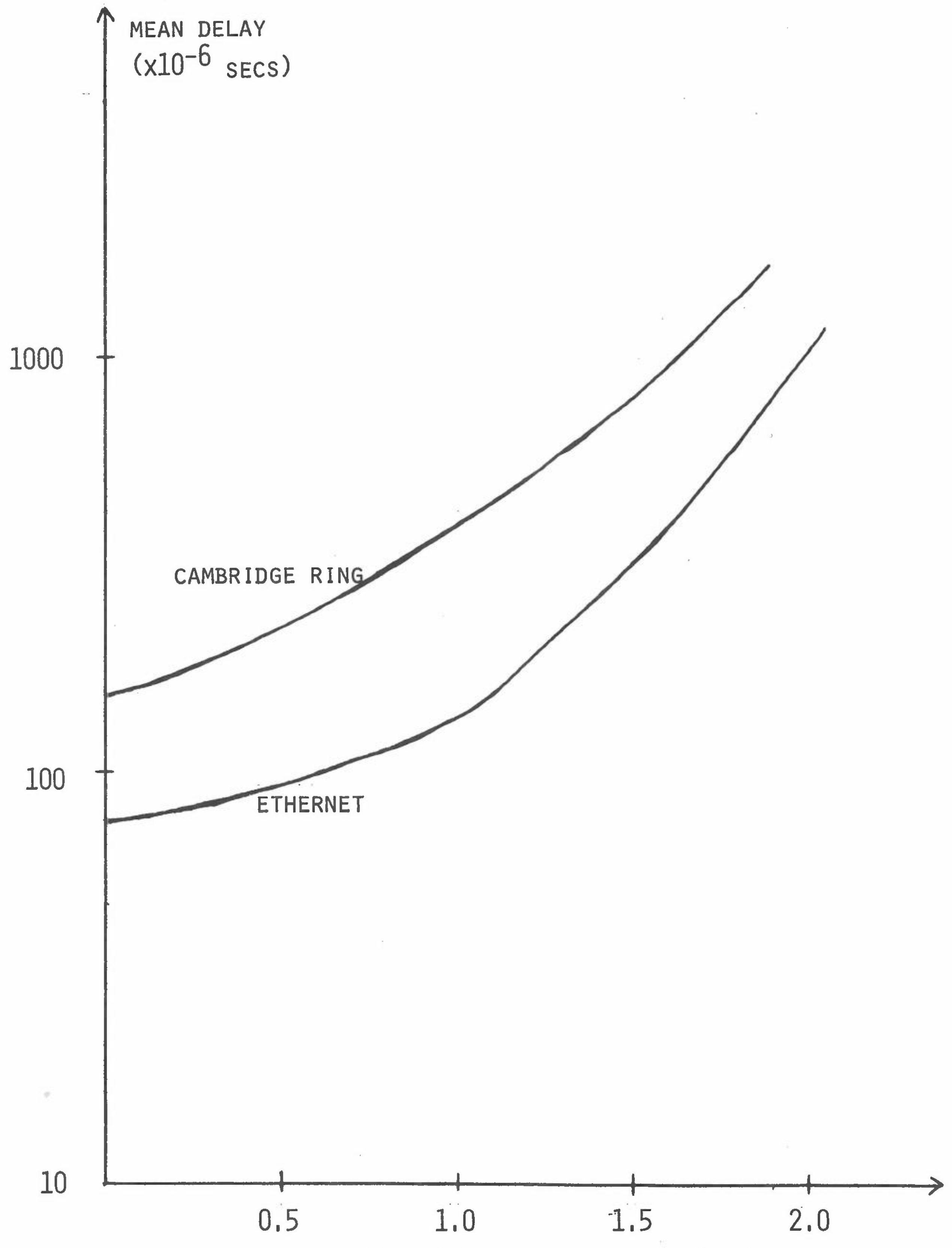
Ethernet Packet



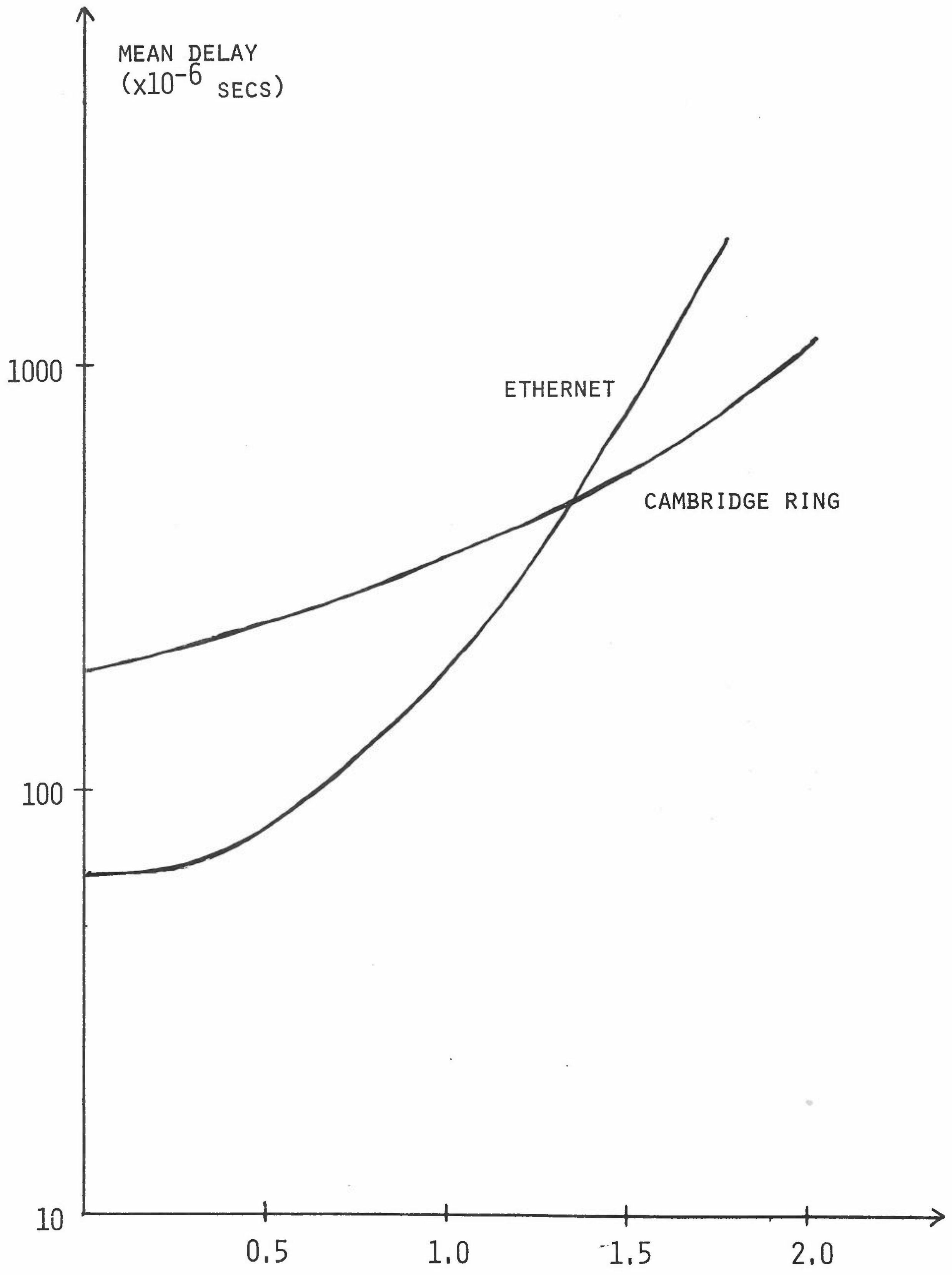
Block Protocol



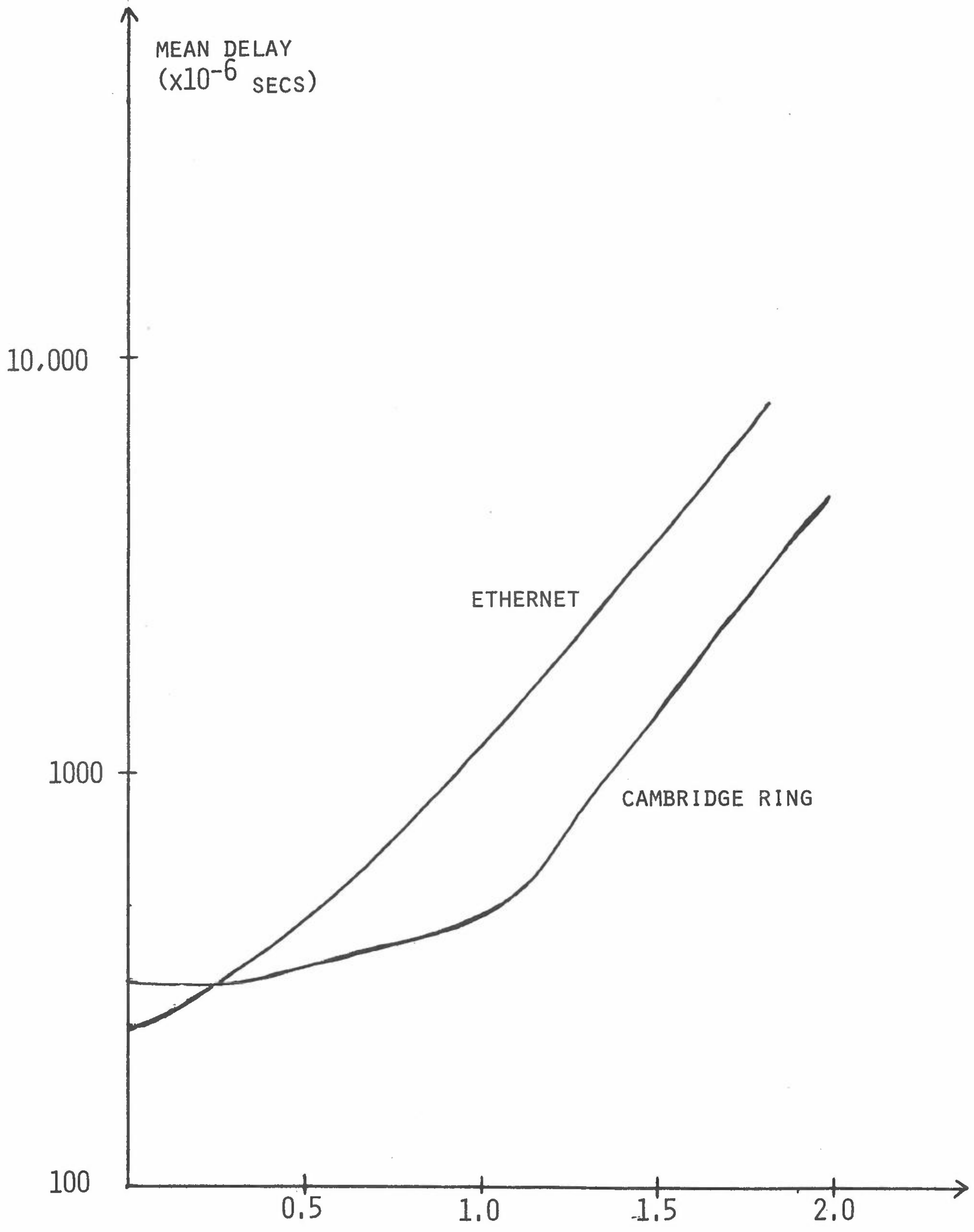
4 STATIONS / 16 BYTE BLOCKS



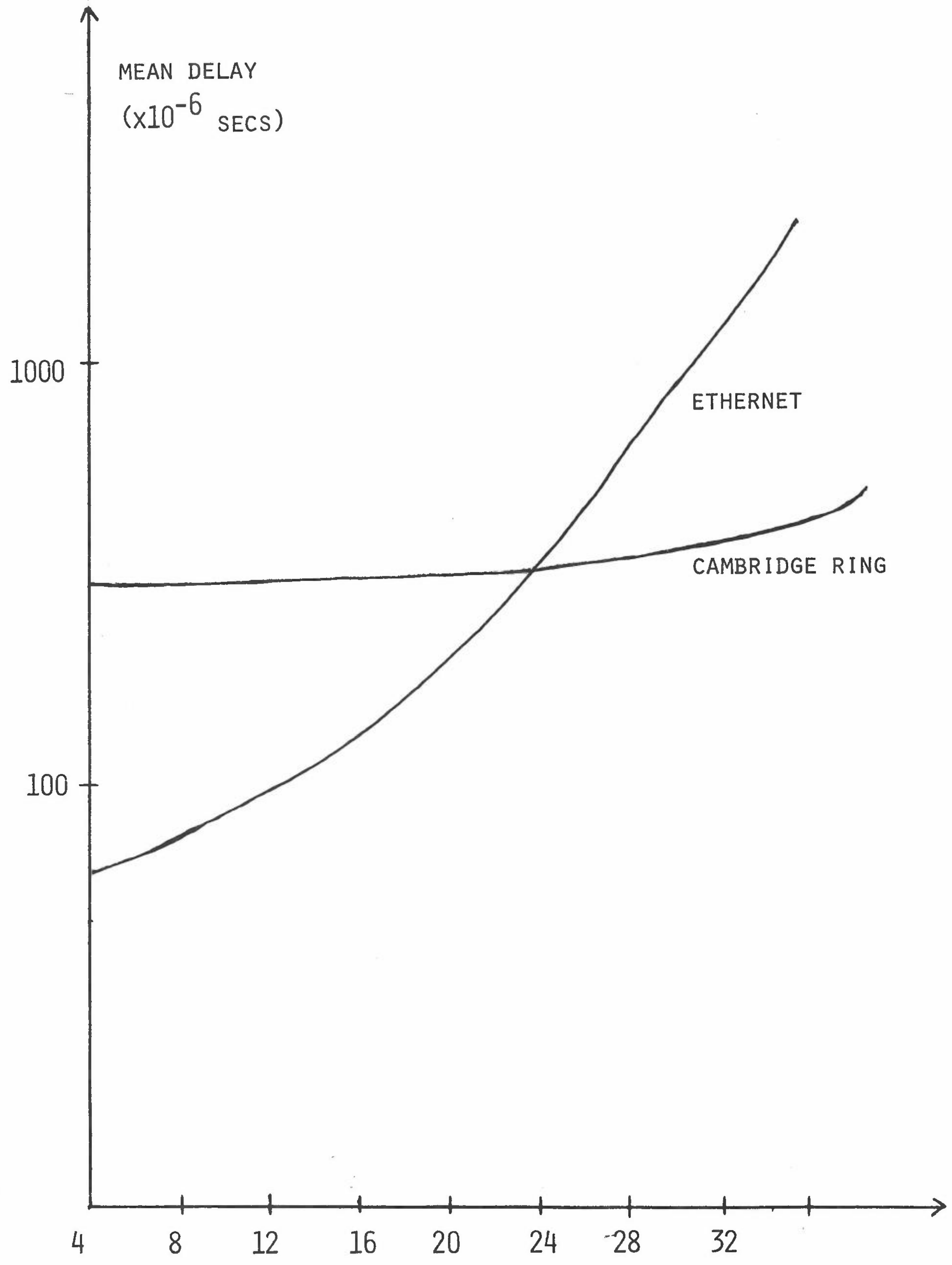
8 STATIONS / 16 BYTE BLOCKS



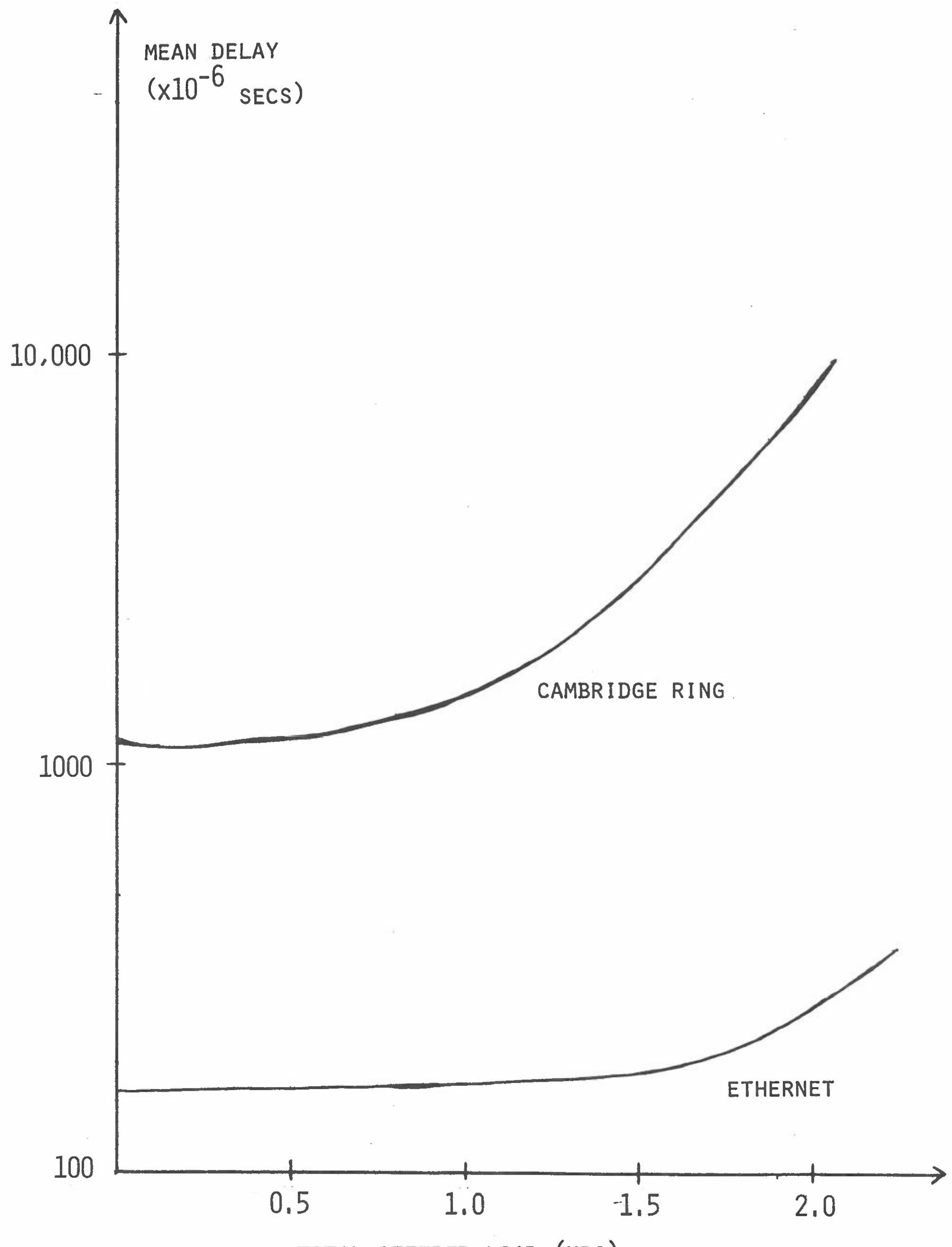
16 STAIONS / 16 BYTE BLOCKS



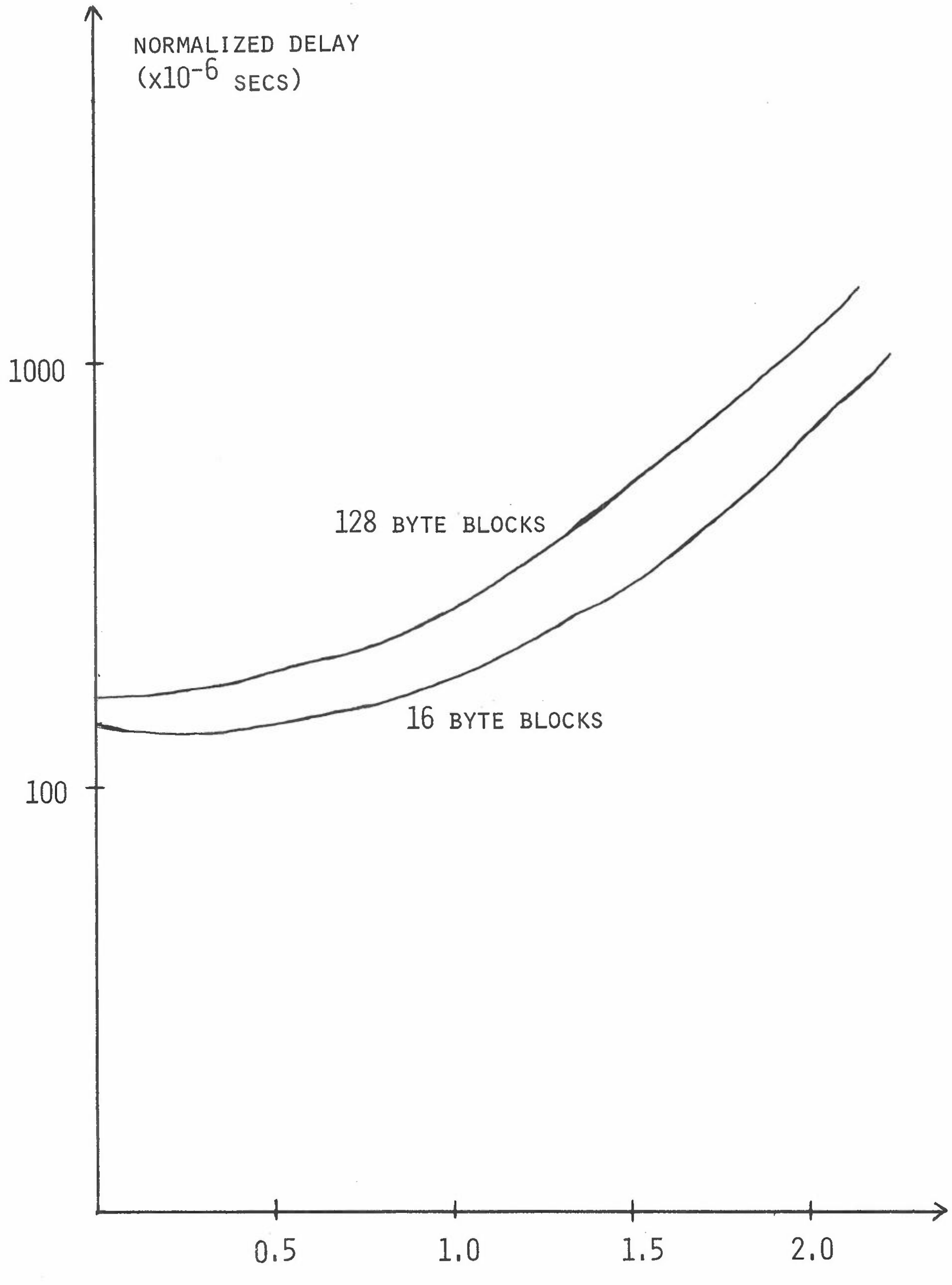
32 STATIONS / 16 BYTE BLOCKS



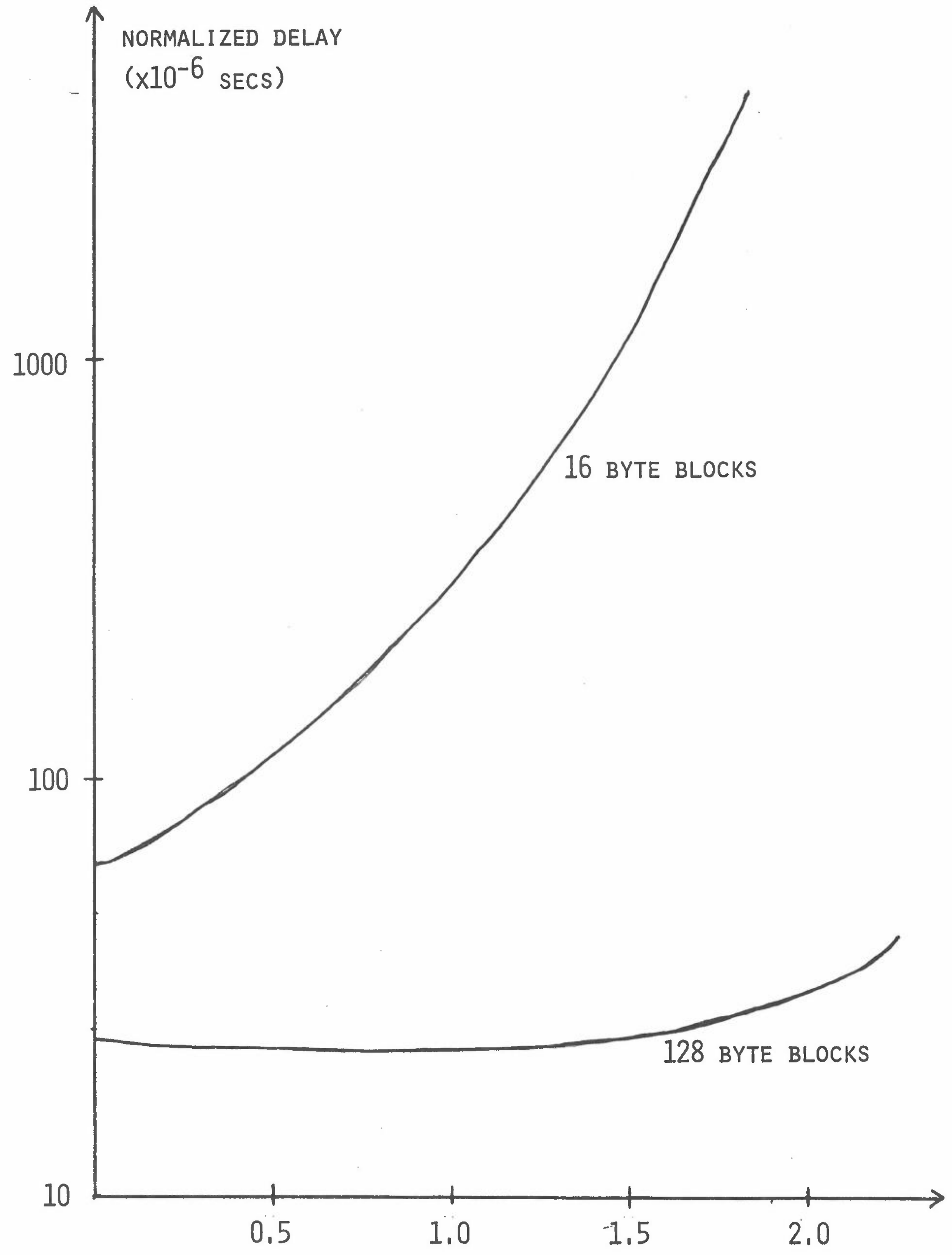
16 BYTE BLOCKS / OFFERED LOAD = 1 MBS



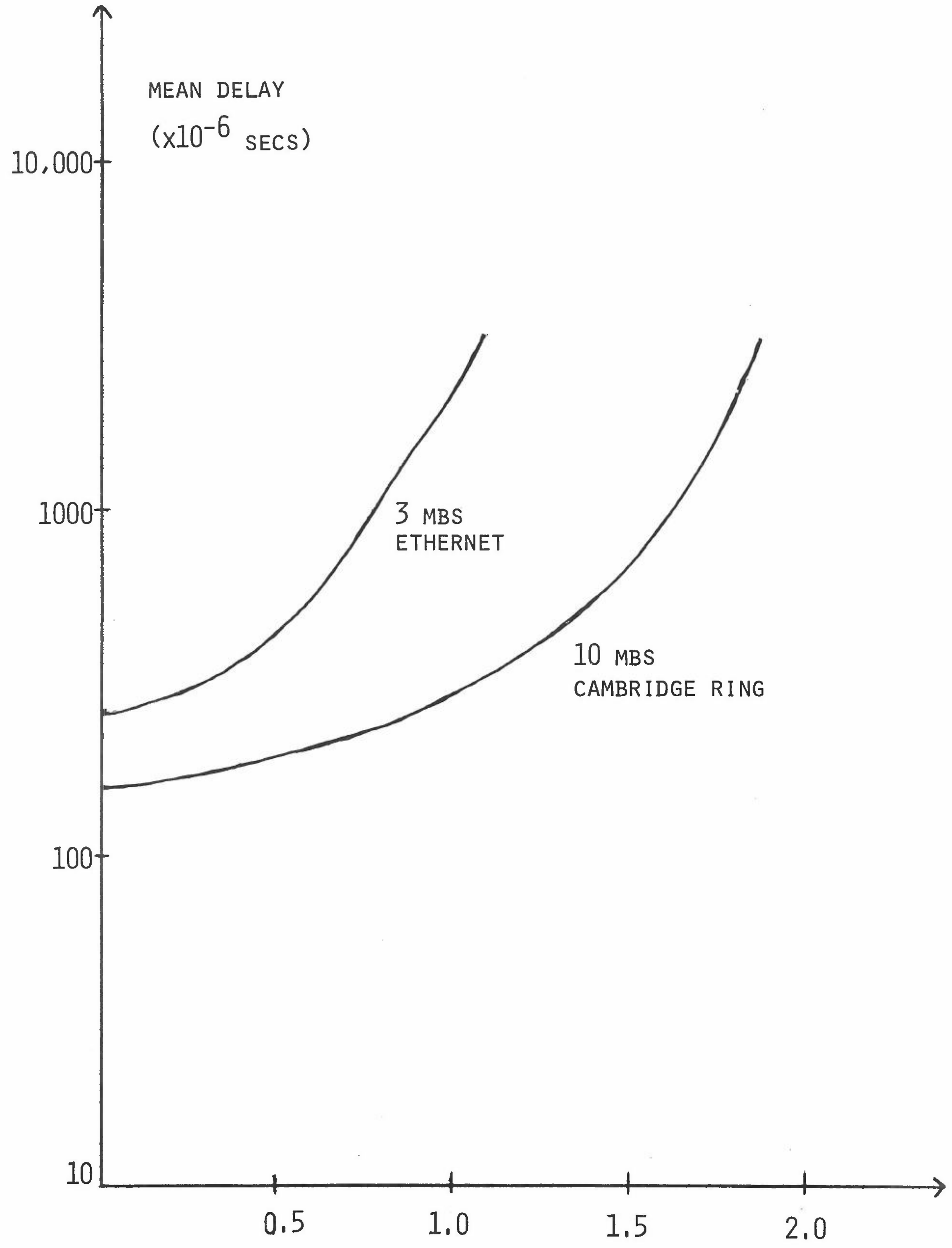
16 STATIONS / 128 BYTE BLOCKS



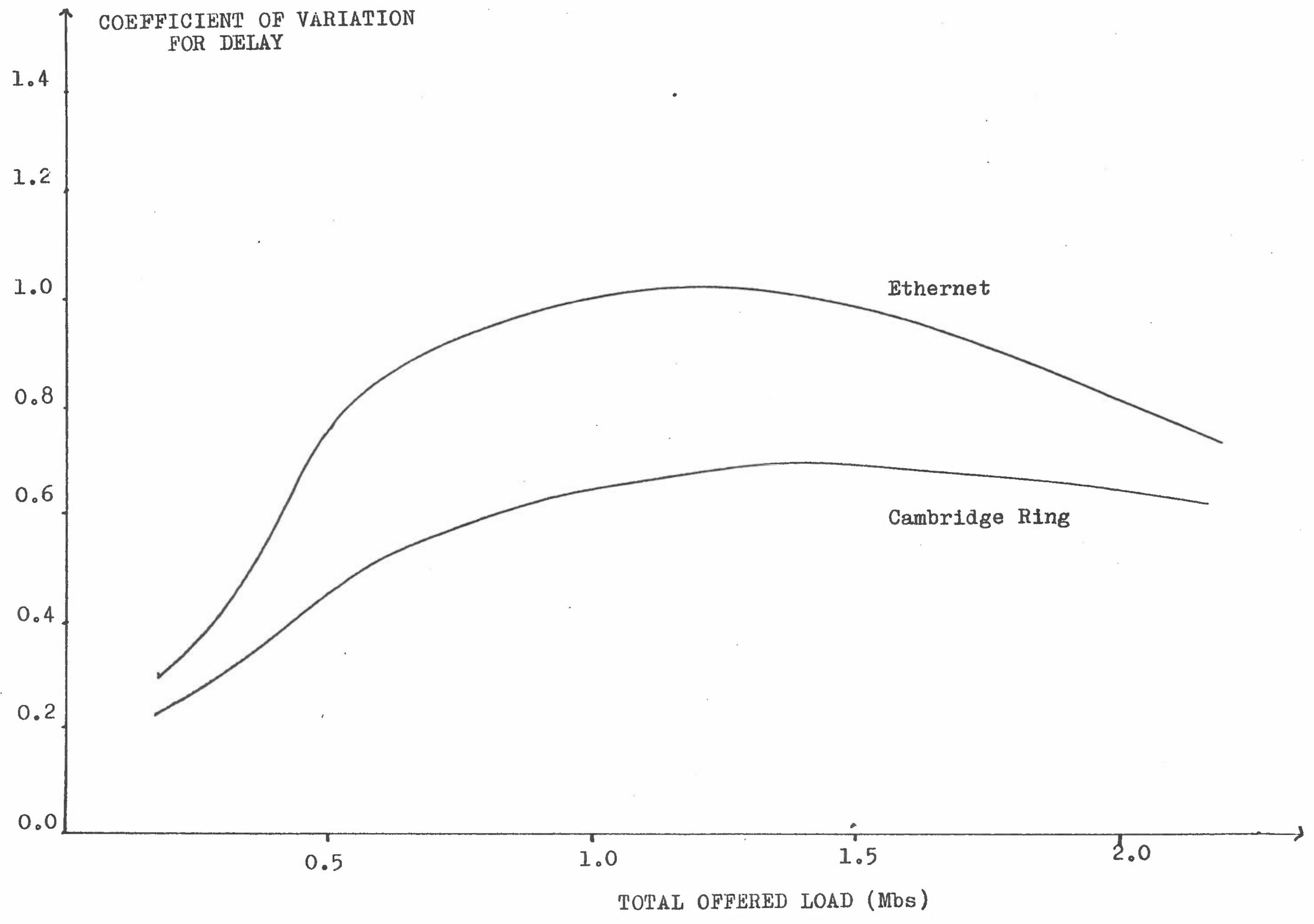
16 STATION CAMBRIDGE RING



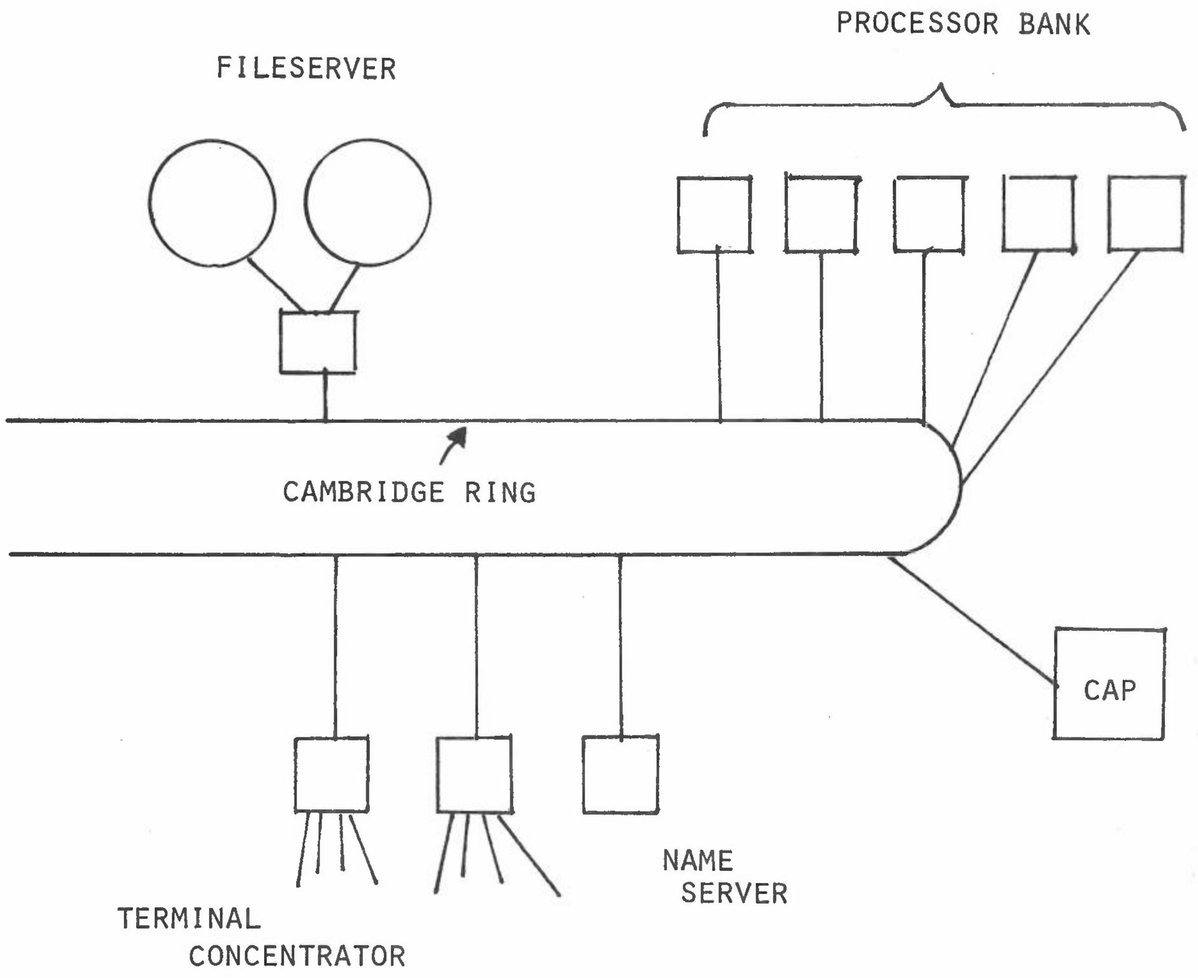
16 STATION ETHERNET



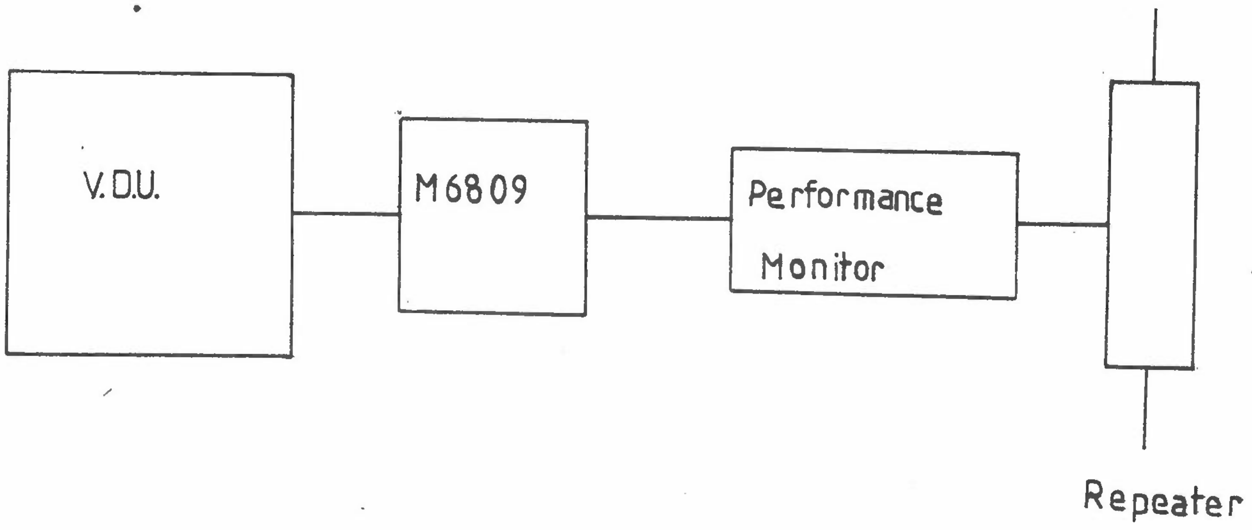
8 STATIONS / 16 BYTE BLOCKS

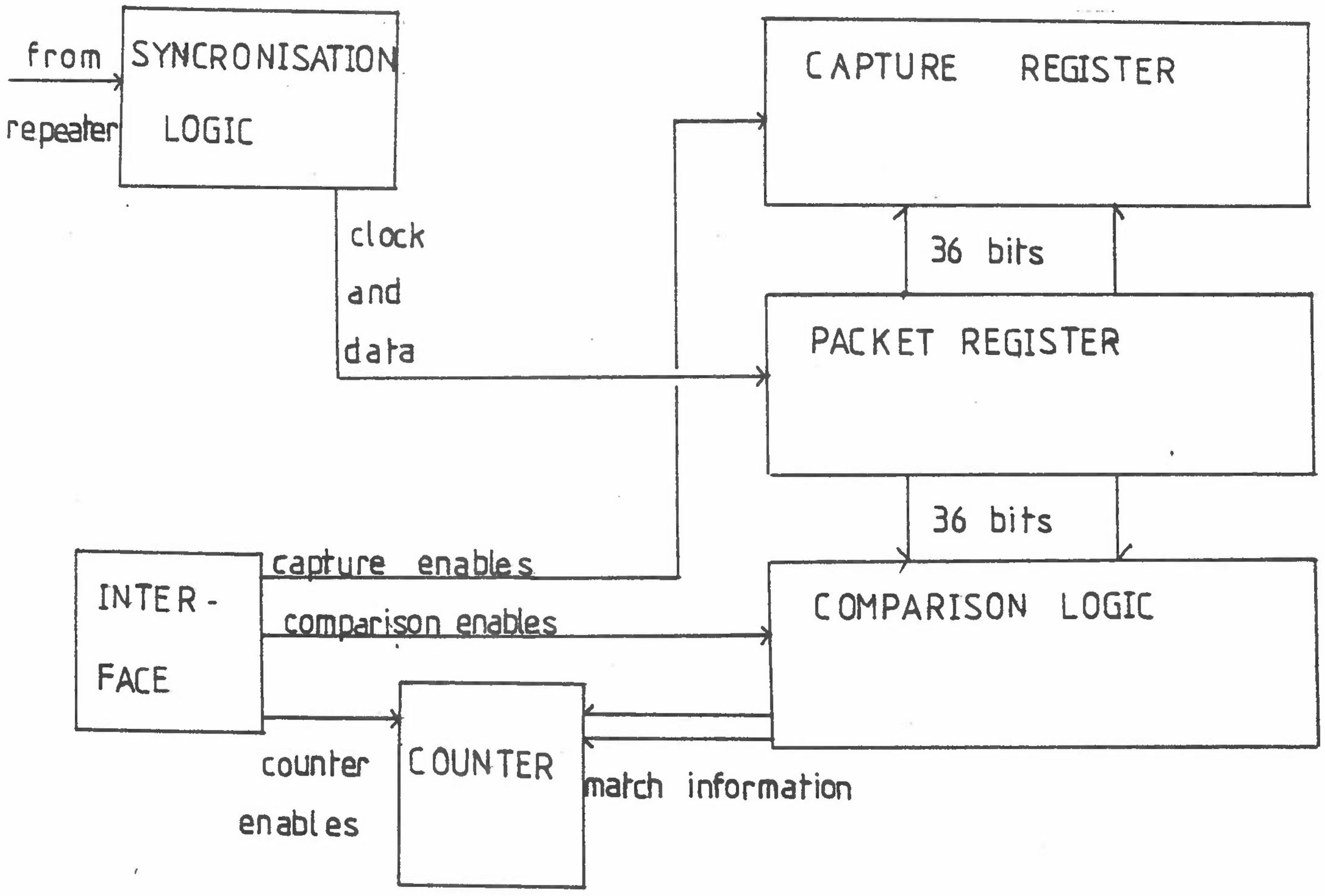


VARIANCE OF DELAY FOR 8 STATIONS / 16 BYTE BLOCKS

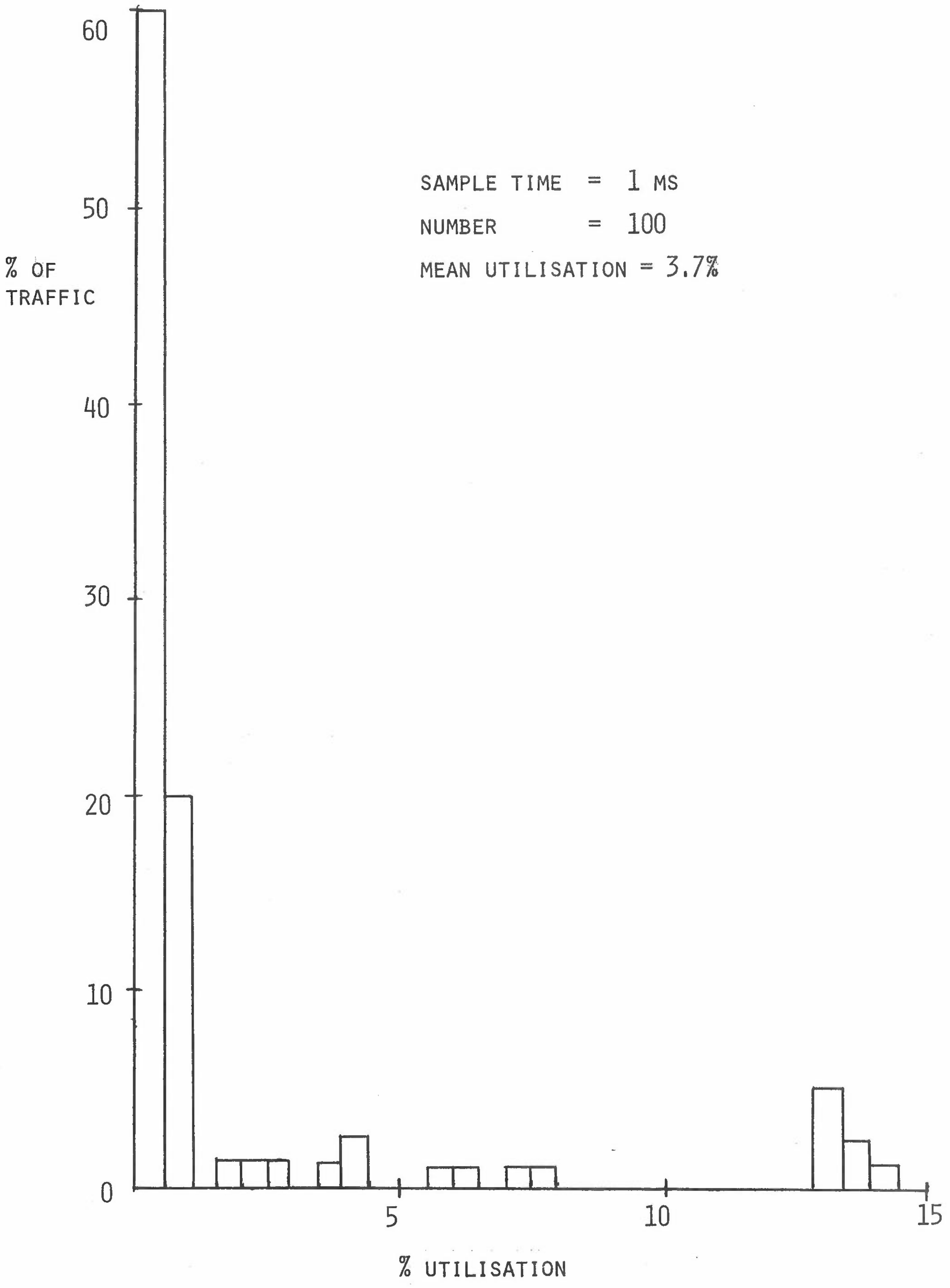


SECTION OF THE CAMBRIDGE DISTRIBUTED COMPUTING SYSTEM

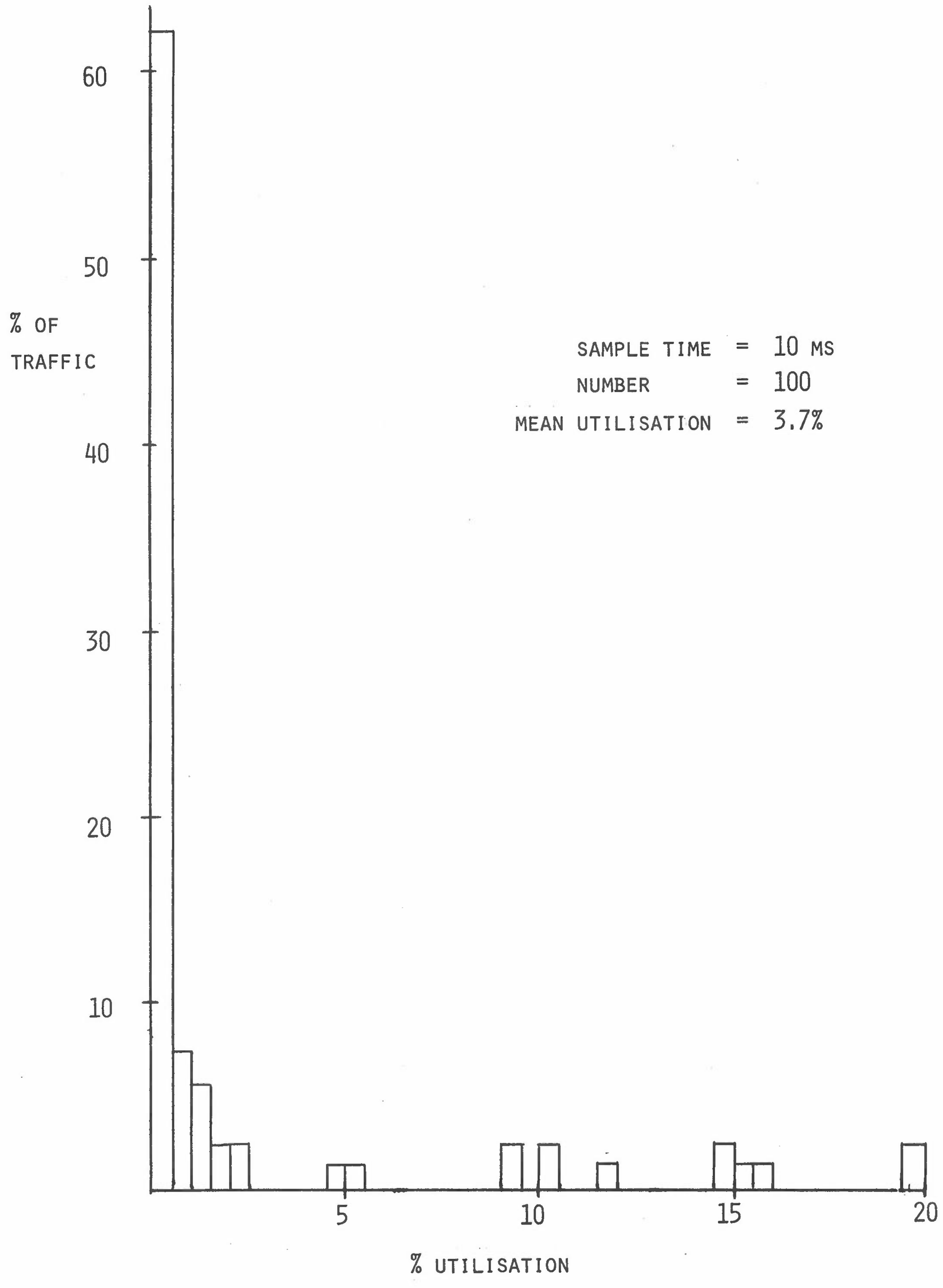




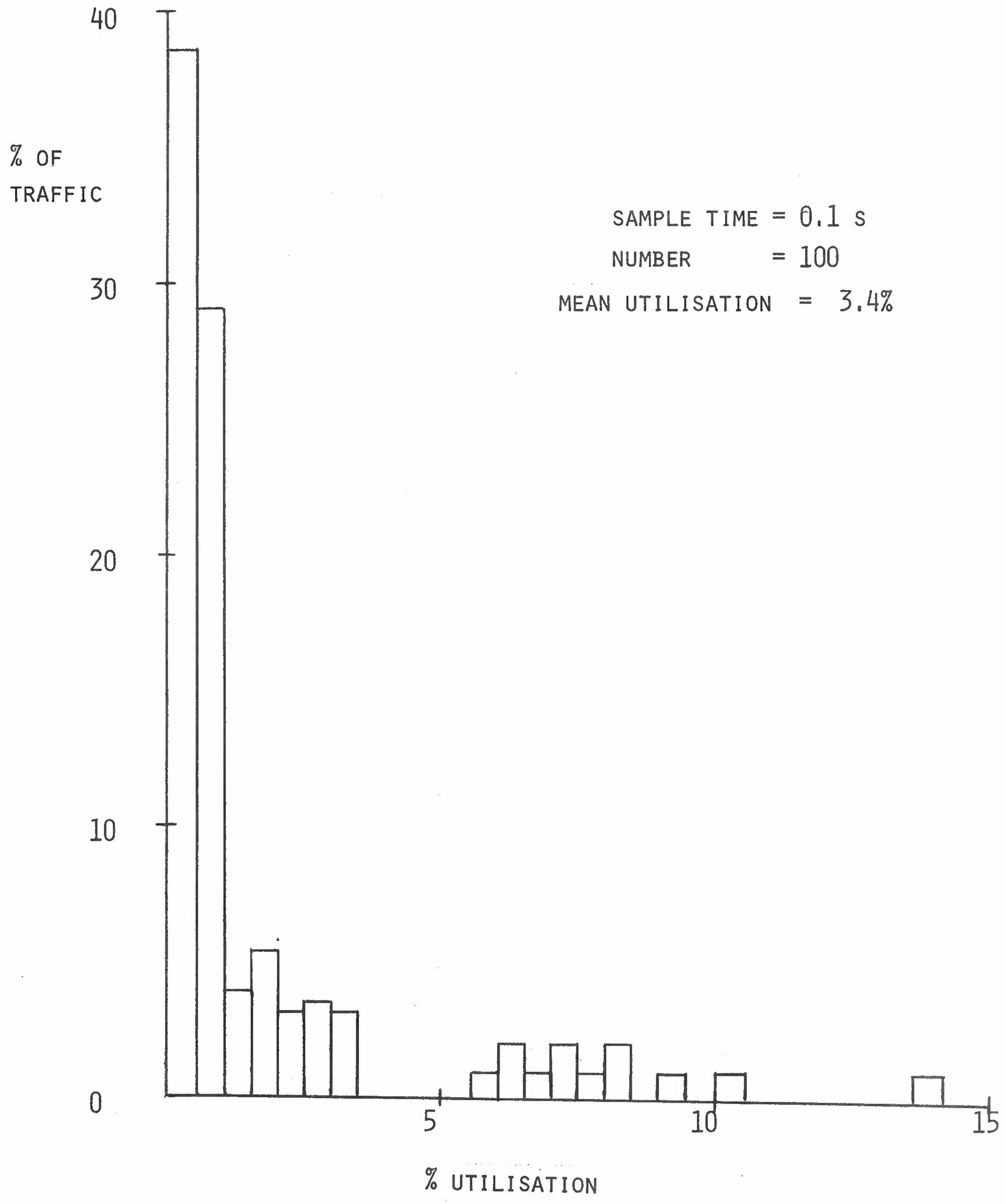
BLOCK DIAGRAM OF PERFORMANCE MONITOR

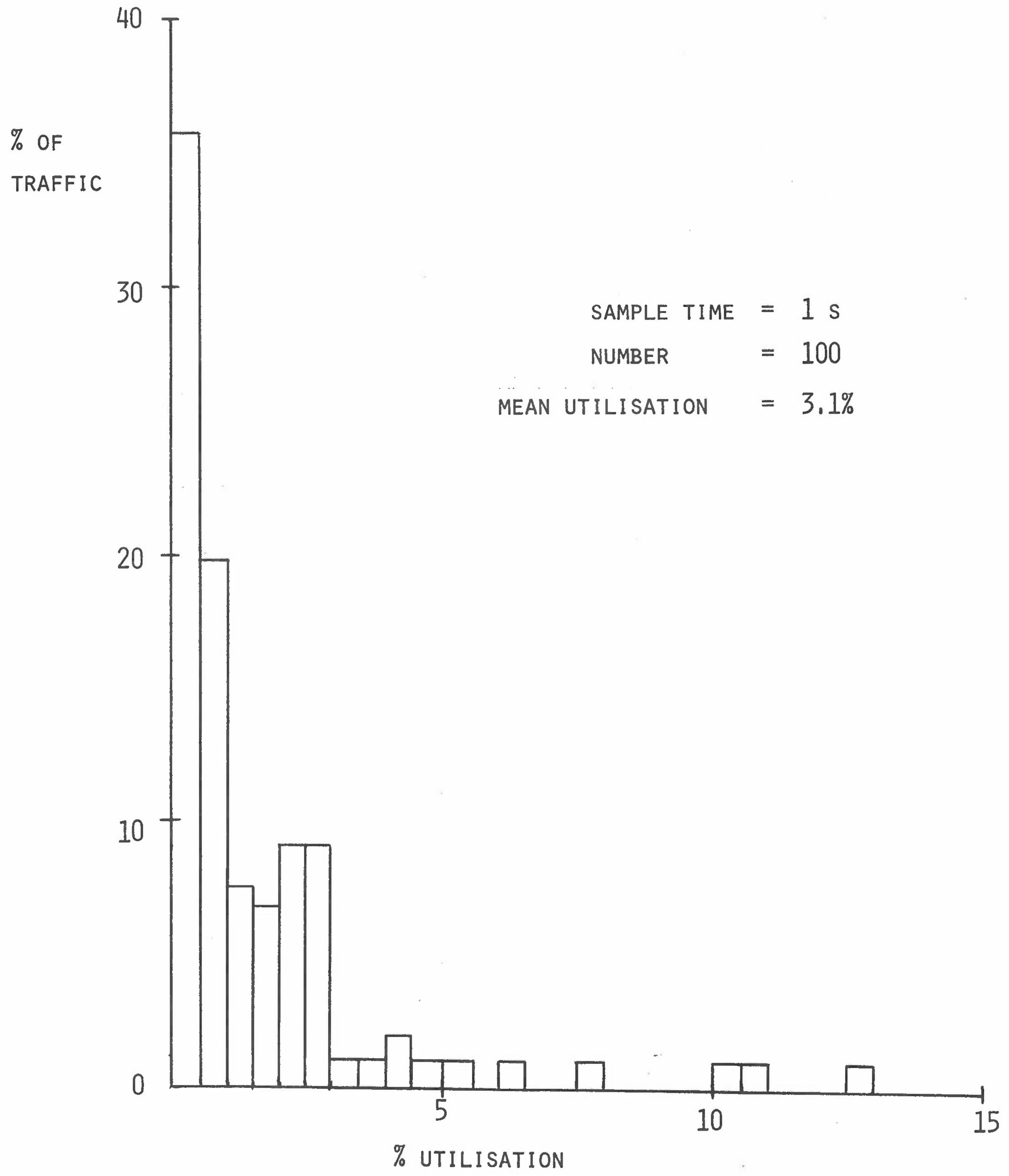


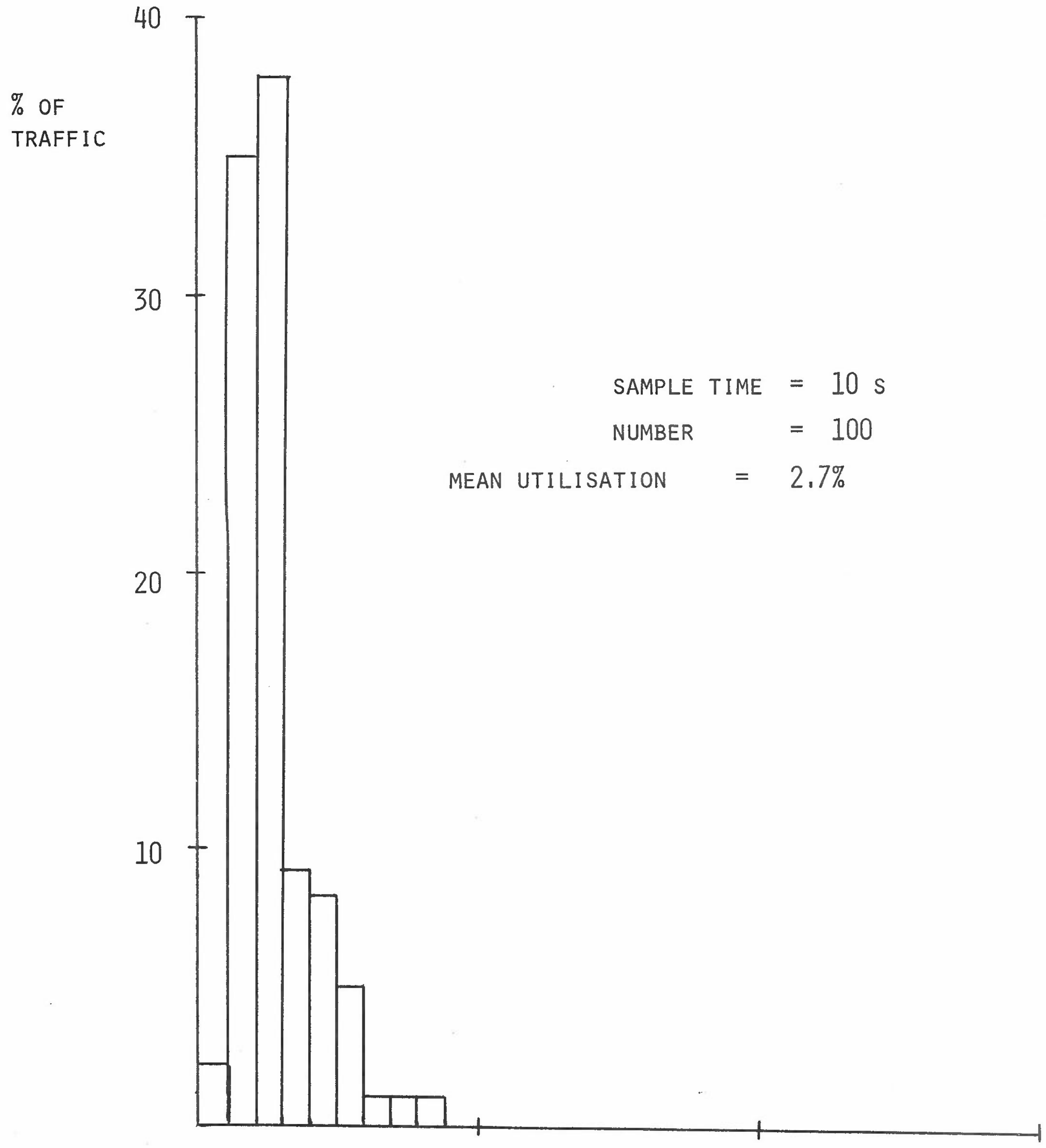
60
50
40
30
20
10
0
% OF TRAFFIC
SAMPLE TIME = 1 MS
NUMBER = 100
MEAN UTILISATION = 3.7%
5
10
15
% UTILISATION



60
50
40
30
20
10
% OF TRAFFIC
SAMPLE TIME = 10 MS
NUMBER = 100
MEAN UTILISATION = 3.7%
5
10
15
20
% UTILISATION


40
30
20
10
0
% OF TRAFFIC
SAMPLE TIME = 0.1 s
NUMBER = 100
MEAN UTILISATION = 3.4%
5
10
15
% UTILISATION


40
% OF
TRAFFIC
30
20
10
0
5
10
15
% UTILISATION
SAMPLE TIME = 1 s
NUMBER = 100
MEAN UTILISATION = 3.1%


% OF
TRAFFIC
40
30
20
10
SAMPLE TIME = 10 s
NUMBER = 100
MEAN UTILISATION = 2.7%

MININET:

AN ULTRA-TRANSPARENT LOCAL AREA NETWORK FOR SERVICE TO HIGH-SPEED INSTRUMENTATION USERS

G.D. Cain *

G. Neri +

R.C.S. Morling *

M. Longhi-Gelati +

ABSTRACT

MININET is a LAN designed to provide attributes and services particularly suitable for instrumentation environments such as are encountered in laboratory automation and industrial process control applications. Most of the obstacles confronting would-be users of other LANs are absorbed into the internal communication subnet, permitting users of MININET to carry on data interchanges without any special concessions to the network's presence. A virtual connection between any two sorts of digital data device (transducer, instrument , or computer) can be established and the extremely short trans-network delays, so essential to real-time monitoring and control, provided. A large, heterogeneous population of device types can be flexibly and reliably interconnected. Low-speed prototype MININETs have been constructed and evaluated; a report is given on the project's progress to date and plans to launch user trials of full-performance network elements during 1983.

* Division of Engineering, Polytechnic of Central London, 115 New Cavendish Street, London W1M 8JS, England

+ CNR-Centro Interazione Operatore Calcolatore, Facolta di Ingegneria, Viale Risorgimento 2, Bologna, Italy

1. WHAT IS NEEDED IN LOCAL AREA NETWORKING

Local Area Networks ("LANs") for data communication between various sorts of devices is all the rage at the moment. There is a veritable avalanche of publicity about this wonderful vehicle for interconnecting the vast assortment of cheap computers, terminals, smart instuments, peripheral input/output devices, and specialist user equipment flooding onto the market. A Martian visitor might be reassured by all the noise and be led to believe that local networking is a comfortable reality. But the natives know different: the LAN situation is presently a mess!

Despite a reasonably mature body of analytical technical literature which has grown up around the substantial success of more than a decade of Wide Area Networks (WANs) working over large geographical expanses, and despite the thundering advances in (and plummeting prices of) digital hardware, there is as yet no commercially-available LAN known to us which even comes close to satisfying the real needs felt by most users. More than one aspiring LAN user has recoiled in dismay upon learning that he cannot easily do for his streams of sophisticated data something like what he has always been able to do for his lowly voice's messages - pick up a telephone on an internal network, comply with a simple dial-up protocol to connect his instrument with a destination instrument, carry out a two-way conversation observing only the loosest of communication protocols, and then close down the connection with reasonable confidence in his ability to redirect to a new destination the next such sequence of actions with an acceptable likelihood of satisfactory service.

The viewpoint most users of a LAN would want to adopt is, as Figure 1 shows, simplicity itself.

The user wishes merely to plug in his equipment (be it computer, terminal, or whatever) into what may be conceived of as a "socket" dangling from a "pipe" (the LAN) which can be instructed to connect up to any other sort of device residing in a suitable subset of the total community of devices that might be available for connection. The user will value highly the ability to gracefully plug and unplug connections and will be inclined to care as little as possible about the detailed way the LAN accomplishes this; the user simply wants to see such action take place when he commands it. He wants physically remote connections to be indistinguishable from connections to user devices coresident with his own. The

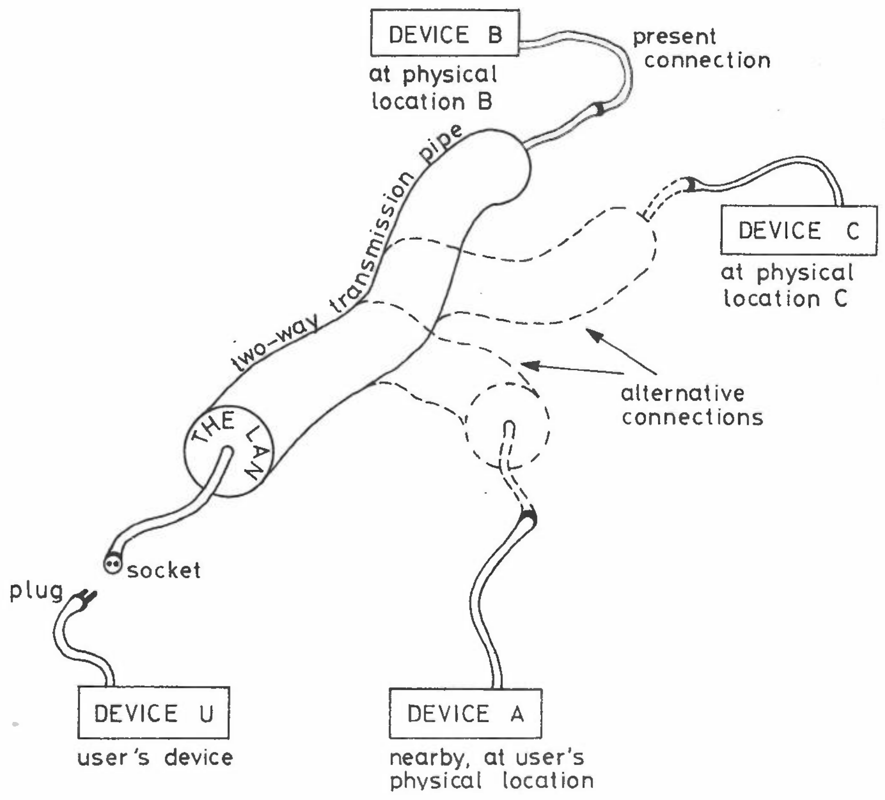


Figure 1. User's Preferred View of a LAN

user wants the pipe to accommodate data transfers having any format and any transmission speed he chooses: that is, the pipe – once connected up – should be transparent. And of course he wants the whole business to be cheap.

It is a recipe for trouble if the LAN builder provides a product that departs much from the user's simplistic view. The LAN builder is well within his rights to impose a standard intermediate interface requirement to serve as the plug/socket arrangement. This is needed even for local connection anyhow, whether or not the LAN pipe exists; there are plenty of successful examples of workable interfaces and some enjoy fairly widespread current acceptance (most notably the IEEE 488 interface [1]). Furthermore, it is physically undeniable that more distant connections will incur greater transport delays in pumping the data from one end of the pipe to the other. Naturally the LAN builder will try to minimize the transport delay and would wish to have an "effective pipe length" which shrinks when physical connections are moved to physical locations closer to the user's location. For the limiting case of connection through the LAN to a device at the very same location ("Device A" of Figure 1), the user must feel that the potential advantage to be gained from unplugging from the LAN pipe and plugging directly into the local device would be so slight as to be overwhelmingly outweighed by the operational flexibility the pipe offers for future connection patterns.

Some users will, of course, want something more than the bare essentials Figure 1 sets out. First, users want high-level services. This is a matter for the end-to-end interchanges between devices. Whatever users or pools of users want to do with their devices' data is between themselves and not the business of the LAN, which should properly present itself only as a very reliable, but unobtrusive communication medium, ready for the occasional connection reconfiguration.

There will be nervous users that want multiple pathways between a pair of devices and gregarious users feeling compelled to broadcast to more than one other device. These special scenarios can be handled by the LAN appearing to present a bundle of pipes into which user devices can wire multiple plugs if desired; but it is still at user level that decisions must be made as to whether the device actually activates operation over more than one pipe.

Users of WANs are typically very different from the potential users of LANs and their needs are in some respects much easier to meet. WANs might, for example, be carrying traffic between computers and faraway terminals dealing in airline reservation or bank account information. LANs, on the other hand, may interconnect devices scattered about a production line or an office complex and might well be supporting traffic relating to automatic manufacturing/testing or stock control information.

The diversity of uses for LANs is expected to be prodigious indeed, and it is hard to arrive at a foolproof categorization of them. Roughly speaking, there are applications in the "leisurely service" category which we mentioned above and might classify as the *office automation* type of LAN, the highly-demanding applications requiring what we could call the *real-time instrumentation* type of LAN needed for laboratory automation/high-speed industrial process control, and of course the gradations between these two extremes.

Office automation LAN users are in many ways spiritually akin to users of existing WANs – they need to convey largish volumes of data among workstations and file servers, but can often live with slow, human-scale response times (hundreds of milliseconds maybe). Nevertheless, the technical problems associated with networks to satisfy this class of LAN user are different in many ways from those to do with WANs, and share broad features with the problems confronting instrumentation LANs. Successful satisfaction of the requirements of the real-time instrumentation class automatically provides a workable solution for the bulk of office automation needs (though the converse is not true), so we will focus hereafter on characterizing this "hot end" of the LAN scene while describing our own approach to serving this sector.

Instrumentation environments often grow up without a comprehensive expansion plan, with a multiplicity of vendors gaining inroads with largely incompatible equipment. While fully-integrated process plants might escape this fate by buying in single-vendor installations all at once and forbidding homebuilt "add-ons" or upgrades employing other vendors' "look-alikes", it is the more usual situation that small user groups in companies and research labs independently accumulate a heterogeneous community of user devices in which some sections eventually outgrow their immediate support resources (in which case resource replication is a possible alternative to sharing of remote resources) or in which incontrovertible necessity for communication arises, say if some resource (like a nuclear reactor, for instance) is absolutely unique. An exceedingly wide range of user device types is likely to be brought into an instrumentation LAN and, in the best voracious maverick tradition, their clamour for unrestricted support will severely test any LAN.

2. THE STATE OF PLAY IN LANs

Three of the scary obstacles confronting a purchaser of electronic equipment are: technical possibility, cost, and standardization. After finally learning that a product he desires is feasible, manufacturable, and affordable, he is faced by the spectre of a dead-end committment if new equipment is "frozen out" by subsequent industry trends toward some incompatibile standard. Fortunately a fairly safe condition prevails in the interface game at the moment, in that the IEEE 488 (or IEC 625 [2]) standard is a comfortable and much-accepted way for connecting instruments and computers.

IEEE 488 is a blessed relief from the RS 232 "standard" that snarls up many a good multi-computer site and, in addition to point-to-point communication possibilities, offers excellent multi-drop capabilities for clusters of devices. However, the communication furnished is of the close proximity kind, with no provision for runs on the order of a few kilometres, which can easily be found in distributed instrumentation systems.

Longer distance (though still "local") communication is instead the province of the LAN. It is now the fashion to speak of data networks in the ordered framework promoted by the ISO Open Systems Interconnection initiative. The "OSI Reference Model" [3] is a handy vehicle not only for discussing and comparing the structures of proposed networks, but also for systematically unravelling the complexity of a network so that its specifications can be partitioned and standards can be established for portions of the complete system. The ideas of the OSI model have propagated widely and can be found described in some books [4],[5] as well as in numerous technical publications and popular press items.

In essence, the OSI model stratifies networks into seven layers which work from the lowest level of physical connection right up to the user's applicication. While manufacturers are busily launching products into the field, standards bodies are feverishly hammering out agreed rules by which "peer entities" (residing generally in separate equipments) at each of these levels can be put into communication. The hierarchy is set up so that a layer provides a *service* to the layer above, making use of the service provided to it by the layer below, enhanced by its own in-layer *protocol*. These services allow (horizontal) peer entity-to-entity protocols of communication to be sustained. Layers communicate (vertically) across *interfaces*.

At the time of writing, the conceptual interfaces and services between Level 1 ("Physical Layer") and Level 2 ("Data Link Layer") and between Level 2 and Level 3 ("Network Layer") have been largely settled. The services required by Level 4 ("Transport Layer") are receiving intense attention by a variety of international standards organizations and advisory groups.

Meanwhile networks are in operation in advance of the existence of any standards. Retrospective mapping of designed and functioning networks onto the OSI model sometimes reveals [5] that independent layering approaches have in fact been adhered to in design practice (which usually is a good thing), though which neworks will prove to comply with any of the emerging standards remains problematical.

Valuable practical work at Layers 1 and 2 is being done, with Ethernet-like systems [6] and Cambridge Ring-like systems [7] commanding the field. However, these developments concern themselves with the lowest two layers, leaving the higher reaches of the OSI layering as the responsibility of the user. The crippling need for heavy user awareness of, and involvement in the operation of, the functions of the Network Layer precludes the carefree style of usage described in conjunction with Figure 1. The upshot of this is that the poor user faces stringent restrictions on transaction speeds, connectability, data format, and procedures. Plus, he normally has to provide and adapt his own general-purpose host computers to take part in specialized network-related jobs!

There are very few LAN developments concerned with the instrumentation sector and, so far as we know, nothing apart from our own work underway which is aimed at directly incorporating the Network Layer features as an inherent part of the LAN. MININET is a true network (in the sense that communicating nodes are capable of transferring information via one or more intermediate nodes) for instrumentation applications, unique in the Network Layer services which are being built in. The firm intention is to liberate the user from the confines of network awareness and to make Figure 1 a reality.

3. FEATURES OF MININET IN BRIEF

MININET is a LAN based on packet-switching technology which is primarily designed to satisfy the requirements of instrumentation environments. A short list of its features includes:

* Ability to interconnect a heterogeneous population of user devices
* Caters for non-intelligent user devices
* Choice of arbitary interconnection topology
* Multi-media links
* Paves the way for easily-redirected resource sharing

* Supports very high communication speeds
* Ultra-high transparency
* Express treatment of short messages by means of word switching
* Flexible yet reliable

All these attributes taken together constitute a powerful aid to users and to providers of LAN services. Elaboration on these features and why most users will value most of them is now given.

4. THE USER DEVICES THEMSELVES

MININET is not a computer network per se, since there is no expectation that a device directly serviced by the network have any computational capability or bear any connection to a computer. Many humble devices will fill this bill; for instance, we may wish to digitize a transducer voltage with an A/D converter and pass the stream of readings across the network to a distant D/A converter feeding one channel of a multi-channel pen recorder. At this destination other readings from scattered sites might also contribute recordings for human assessment and action.

While MININET users are under no obligation to hook any sort of computer onto the network (since all computational power required for operation of the network is incorporated directly in the elements of MININET) there is, of course, likewise no reason not to make every device connected via the network a computer of some size or description. If so, we are merely landed with a special (computer network) case. Sometimes it is desired to pass all data from transducers, converters, and instruments first through a proximate host computer for digestion, prior to onward transmission. Whether computers get in on the act or not (and surely the embedding of cheap microprocessors in much equipment increases the likelihood that they will) MININET is unlike many other LAN solutions in that it sets no intelligence barriers. Any device dealing with digital data is welcomed with open arms.

5. TOPOLOGY, NETWORK ELEMENTS, LINK TYPES AND RESOURCE SHARING

It is often true that a pool of small computers needs access to a pool of peripheral devices such as line printers and plotters. Rather than spend a fortune on attaching dedicated (under-utilized) peripherals to each mini and microcomputer it is usually sensible to try a strategy of sharing. The connectivity maze explodes if many computers and devices are included in the scenario, so some have sought solution by the dreaded centralization of resources, with interconnections limited to vulnerable star, bus or ring topologies. MININET allows any topology to be chosen. Its nodal elements can be snapped together through point-to-point links in any desired layout, as Figure 2 exhibits. Internodal

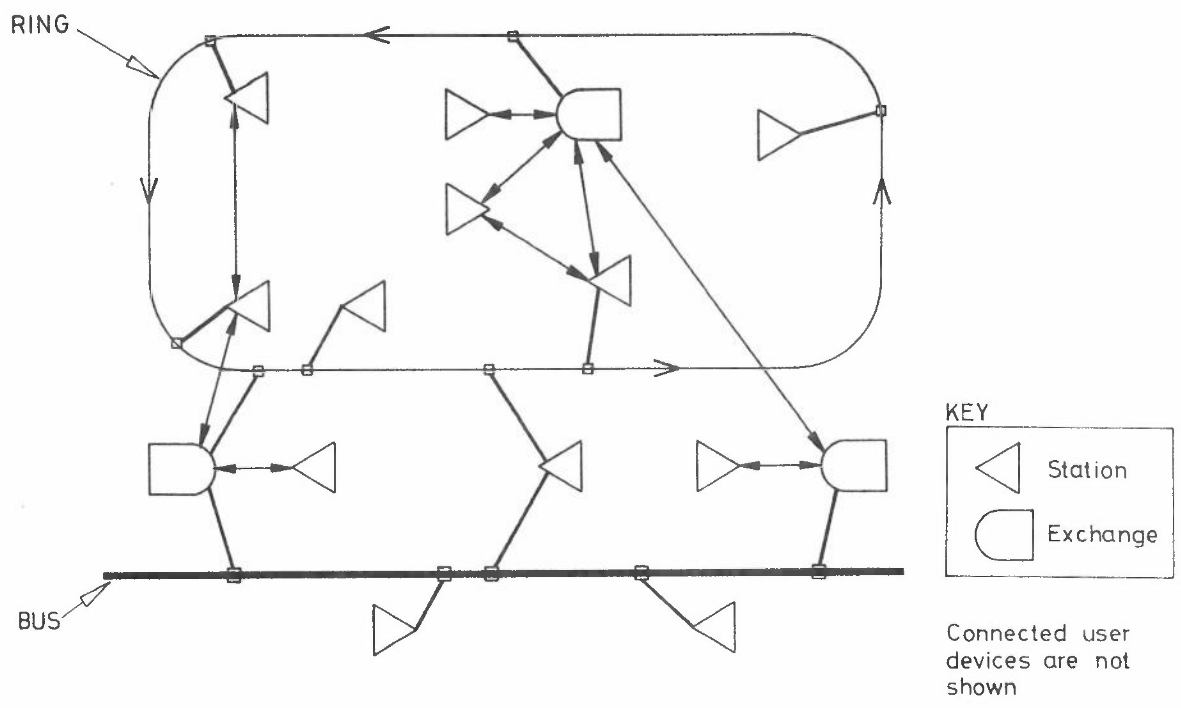


Figure 2. An Example MININET Showing Possible Connectons and Topology

distances can be up to about one kilometre.

In looking at Figure 2, the reader must supply from his imagination a great number of user devices sprinkled about, since their explicit inclusion would hopelessly clutter this drawing. Over 900 user devices could be supported by the example arrangement in the diagram alone, and this is less than a quarter of a single MININET's capacity!

Figure 2 shows the two main network elements: Stations and Exchanges. A *Station* is a network node which provides the user access into MININET. An *Exchange* is a node that provides, in addition, a store-and-forward relay function, routing packets of user information towards their destinations. Each of these sophisticated special processors contains sufficient inherent computational resources to function without reference to any external service or host computer. There can be up to 64 user devices attached to each node and there can be 64 nodes in a MININET.

Each line between nodes represents a physical link which is independent of every other such link. Some links (referred to as MININET *channels*) could be slow (e.g. telephone links) while others could be very fast (say, optical fibre links). This multiplicity of media is nearly unheard of in current LAN practice, despite its obvious desirability if cost or operational factors recommend its adoption. Overall network speed will not be reduced to the speed of the slowest of the links, as happens with non-homogeneous rings.

In addition to point-to-point links MININET can cheerfully utilize multi-node channels such as rings and buses, as the intermeshing in Figure 2 clearly demonstrates. In other words, MININET is capable of providing its Level 3 service by utilizing ring (Cambridge Ring, for example) or bus (Ethernet, for example) Level 2 structures.

Returning to the issue of resource sharing, it is evident that a great many dispersed devices constitute the resource community. Each user device can be allocated a *virtual connection* through MININET to another device, on a one-to-one basis. This is what Figure 3 indicates. Blissfully Figure 3 hides the complexity of the detailed physical pathway used to connect devices. It is obvious that the ability to easily redirect data flowing on the Device U - Device B virtual connection to a Device U - Device C virtual connection will be a valuable asset and will, among other things, greatly facilitate resource sharing. This can be done in MININET: a user either instigates human intervention to the operator's console or through a management Port connecting in a cooperating host computer which can act as a reservation agent. In any case, a "dial-up" sort of request is issued from Station 6. If the relevant network management entities agree that a disconnect on the Station 6, Port 5 - Station 4, Port 3 virtual connection can be completed in

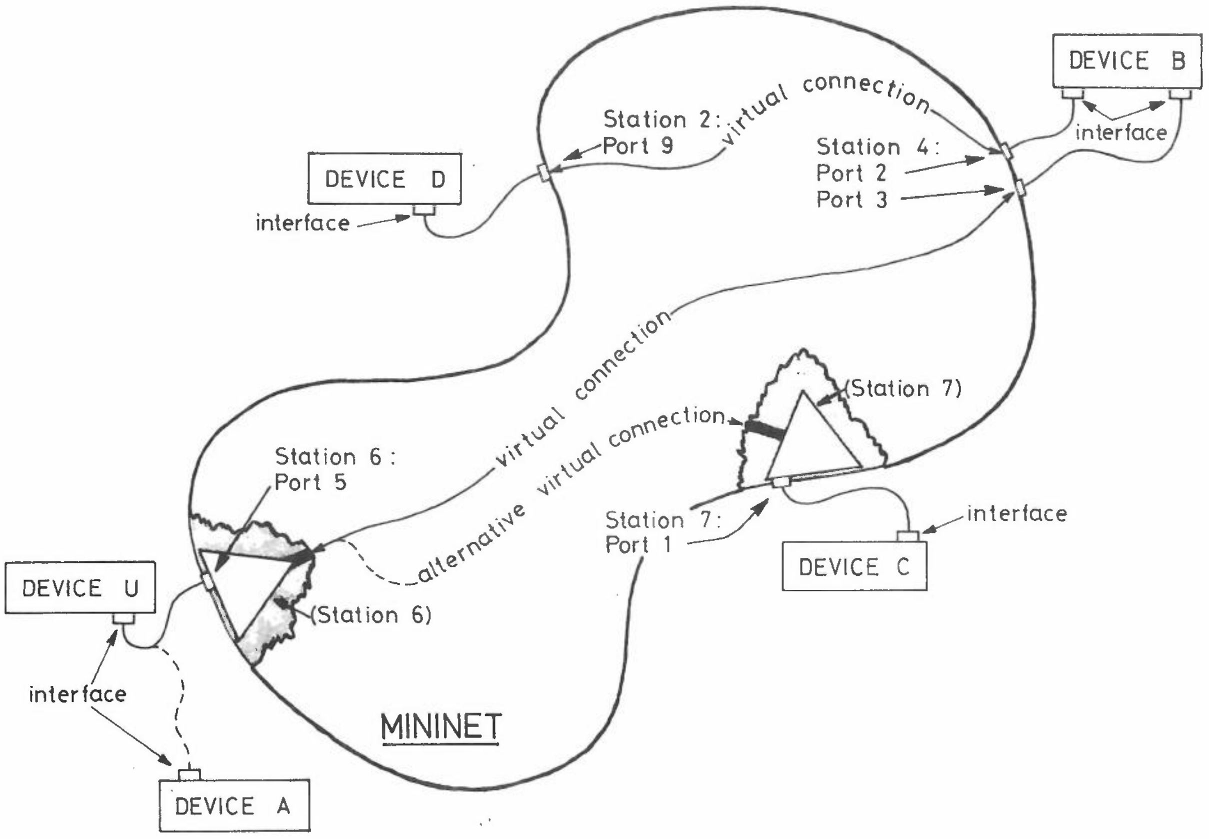


Figure 3. How MININET Becomes Like Figure 1

favour of the virtual connection shown dotted to Station 7, Port 1 then this is carried out automatically.

We have found this flexibility of use to be a great boon to MININET users.

6. SPEED, TRANSPARENCY AND THE MESSAGES

The term "real-time" is highly subjective and its popular computer connotation signifies very slow stuff compared to the needs of instrumentation. Instrumentation LANs must meet real real-time deadlines since the data is typically highly volatile. Many of the user transducers and instruments may have no storage capabilities whatsoever, so there is no option but to flush away the data as fast as possible.

Also it is in the nature of instrumentation and control systems to need very quick delivery of short messages. Suppose Device U is a flotation transducer detecting the imminent overflow of fluid in a tank. If Device B is a computer at a central monitoring and command centre it needs to receive Device U's report immediately - perhaps that warning message would be a single data word. There can be no hanging about to assemble other words from Device U or even other words from the other Ports on Station 6 (the connected devices might be dormant at that time anyhow). Express handling of that single word from Device U could be of paramount importance. Perhaps the computer (Device B) would subsequently issue an equally-important switch-on command word that needs to hotfoot its way through the network to actuate a pump, Device D.

These considerations highlight the differences between this sort of environment and the office automation style of demands. Because of the burstiness of the instrumentation traffic and the volatility of each individual word, the data packets must be short. In MININET, a packet contains one 16-bit user word - so we have a word-switching network. The complete packet conveyed in the network is 32 bits in length. Also the trans-network delivery delay time must be low. Our design goal is to provide transaction speeds of 100 K packets per second (so users get word-switched transactions at up to 1.6 M bit/second)! Even more significant for us than this high sustained throughput figure, is the goal of mean internodal (hop) delays of about 50 microseconds.

The effect of the extraordinary transaction speed provision of MININET (not to be confused with the high line speeds quoted for other LANs) is to roll back the barrier of speed enormously, so that people with slow speed applications can simply view their connections as instantaneous and direct their worries elsewhere. People with very high speed demands (nuclear experimental control, packetized speech processing, etc.) will be able at last to live with networking.

So high speed, by obliterating bottlenecks, promotes transparency. Equally, word-sized packet communication aids transparency. Since the common currencies of the instrumentation world (16-bit words) are being interchanged, one packet carries the equivalent of a single physical transfer between a computer and a device. This extends the concept of transparency well beyond mere code independence. Do-it-yourself protocols at user level are not put into disarray when communication is cranked up using the network.

7. FLEXIBILITY AND RELIABILITY

Considerable care has been taken to ensure that a very highly reliable brand of communication takes place in MININET. Layered precautions in hardware, software and protocols give us cause for confidence that under all but the severest of operational calamaties, users will receive proper sequential delivery of their data words. Heavy network loading and the harsh noise environment of the shop floor may conspire to choke back the network throughput, but users can expect degradation to be graceful and orderly. Inevitable link and node failures should be absorbed as soft, recoverable failures. And, by virtue of a *fairness criterion* imposed at all levels of the network design, no subsets of user devices "hog" network resources, regardless of their state.

MININET flexibility starts by the user being able to ignore the network and believe the fiction that communication is being provided by a direct local connection instead of through the network. If he has adopted the interface required to be compatible with the network he can be oblivious of MININET's presence or absence. For instance, Device U of Figure 3 could be disconnected from Device A and plugged into Port 5 of Station 6, where a virtual connection through to Device B might already be set up. If Device B were identical to Device A then perfect communication could resume (assuming the slightly longer propagation delay caused no trouble at user level). <u>Not one iota of change to the hardware or software would be necessitated by the presence of MININET!</u>

This near-invisibility is alien to mainstream LAN guarantees and provides the solid basis for operational flexibility in MININET. We are sometimes told that the easy incorporation of a microprocessor (for handling network communication) into every possible user device - thereby making each device network-conscious - is a sensible alternative to our rigid requirement for zero user-invasiveness of MININET. On at least two counts this contention is fallacious. First, network-related tasks are complicated and many very simple homebuilt devices would be dwarfed in cost, size and complexity by the system needed to strap onto them. (Recalling that a full-blown MININET could support 4096 user devices, it is outrageous to entertain hardware and construction costs which go beyond the absolute minimum needed - the interface.) Secondly, network-related software modifications and upgrades would present an unmanageable problem. For, even if all such mods could be downline broadcast (more likely, up to 4096 PROM sets would need to be retrofitted), how could a network manager be certain which of his many devices were operational and on-line on the day the changes were piped down? Transparency is not only desirable, it is nearly mandatory. Any user intelligence should be expended only on application-related tasks.

Finally, further MININET flexibility comes in how the network itself can undergo physical reconfiguration during operation. Nodes can be taken out of service or replaced during network operation. User devices will be routinely unplugged or plugged in without affecting the network.

8. A TASTE OF THE DETAILS

Just because MININET will be a simple system as far as the user sees, does not mean that MININET itself is simple. A good team has been at work for a long time tackling the unique problems presented in every aspect of the hardware, software, procedures and protocols. Essentially it has not been possible to adopt anything prefabricated; everything has been built from the ground up.

The work has, from the outset, been characterized by fruitful international collaboration. Part of the team has been based at the Polytechnic of Central London and part at Bologna University. Readers interested in following the detailed technical aspects of the work can be supplied some of the documents which tell the story (e.g. [8] - [14]). The intention of this section of the paper is merely to briefly give a flavour of some of the technical issues.

First, there is the matter of a suitable hardware interface. We have developed, and have successfully used for more than six years, a low-cost parallel interface called DIM [8] which is the mainstay of all our laboratory communication and is the prime interfacing mode for MININET. A less provincial capability will be provided through a Port for IEEE 488 devices.

After defining the full target specification for MININET [9], much of our early efforts went into the definition of two protocols. The MININET Link Protocol (MLP) ensures sequential packet interchanges between nodes under all error and retransmission eventualities [10]. The MININET Control Protocol (MCP) secures internode communication, never seen by the users, which permits effective network management [11]. The overall hierarchy of concerns and service aspects of MININET is best appreciated by referring to Figure 4.

Flow control, effected by means of Back Pressure Flow Vectors [12], and routing (quasi-fixed, using a tree rooted at the destination node so as to minimize the Channel weighted distance from each node to the root [13]) turned into remarkably complex matters due to, among other things, our insistence on

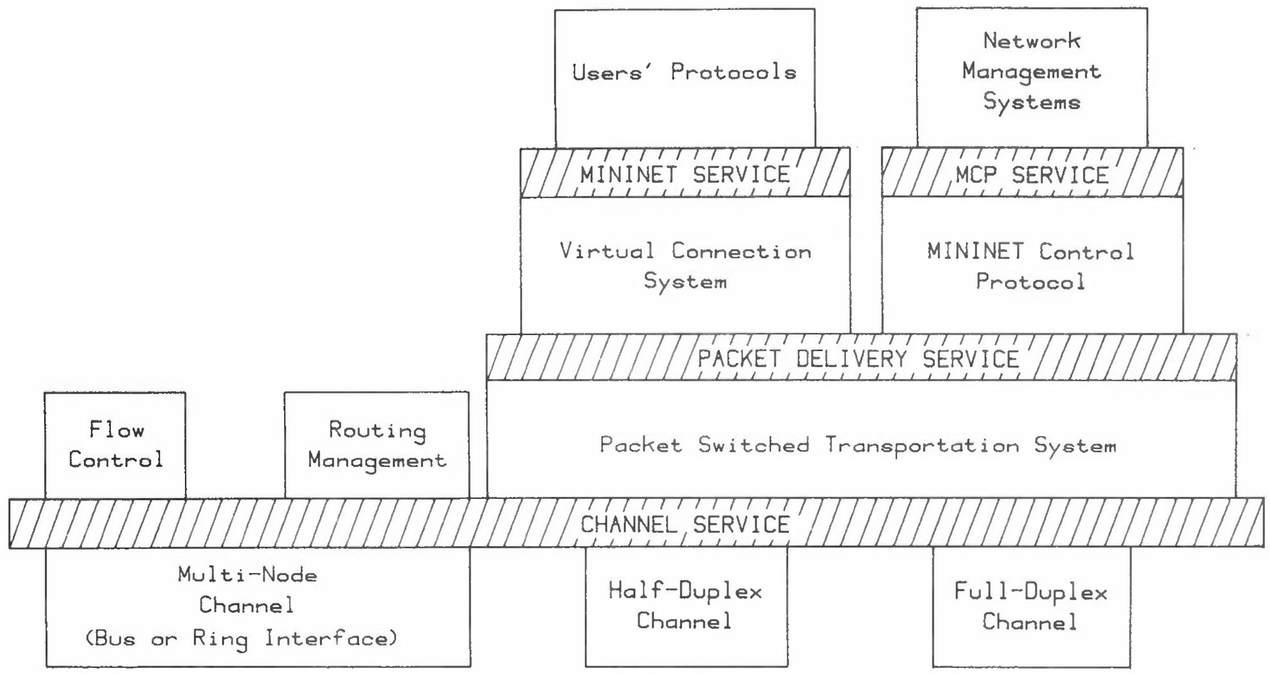


Figure 4. Hierarchical Model of MININET

maintaining intrinsic packet delivery sequentiality from one end of every virtual connection to the other.

The central ideas of MININET were shown to good effect in 1980 when, after an extensive programme of hardware and software development, an operational plateau was reached with a low-speed version of MININET. A good snapshot survey of work up to that point is given in [12]. Stress testing of a small network in London (consisting of four Stations and two Exchanges) exercised and validated the design approaches for speeds of only 30-50 packets per second. Despite the fact that this low-speed design is only a mere shadow of the target system (now coming into service), the performance of a single Station in Bologna acting as a multiplexer for computer room resource sharing has been impressive. The system operates at around 400-500 packets per second, permitting dial-up connection changes between four computer systems, two printers, a paper tape unit and several terminals. Its operation (non-stop since February 1980) is crucial to the operation of that computer room. One five-minute power supply failure and, in late

1982, one printed circuitboard failure caused the only interruptions to perfect service in all that time. It is strongly felt that, as a result of this soak test, there is every likelihood that the target MININET is being well engineered.

The key element in launching the full-specification MININET is the Station, since it is this special-purpose communications processor which gives MININET a big edge over LANs that suffer the bottleneck of a slow, general-purpose processor (user-supplied sometimes) for network entry. The functional representation of the Station looks simple (Figure 5). Taking the lid off to look at

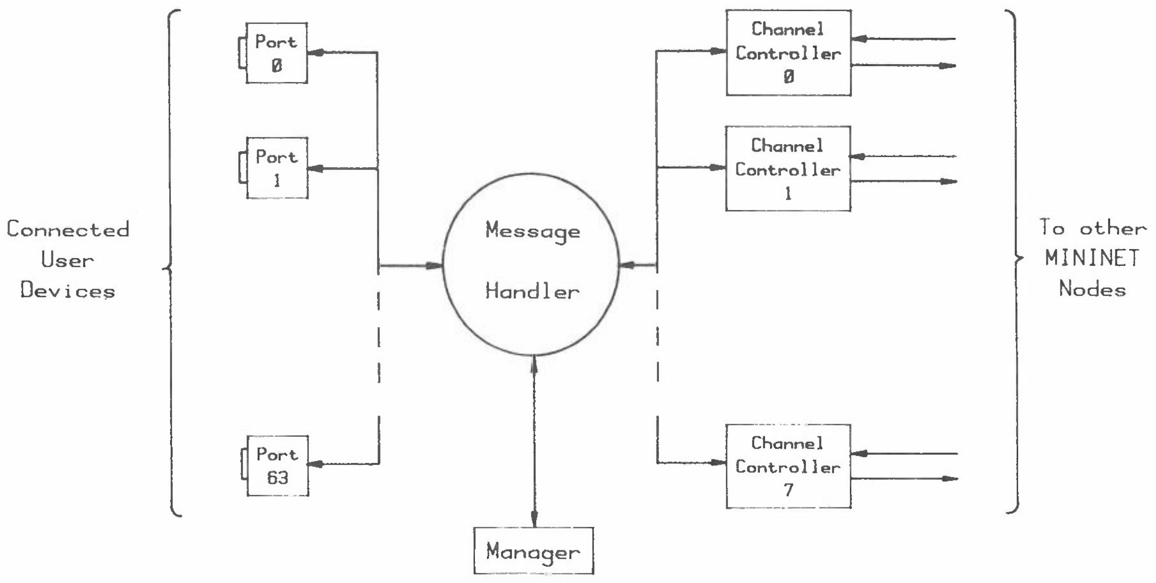


Figure 5. Functional Division of the MININET Station

the structure, though, reveals that it is a complicated beast (Figure 6) which, apart from hardware that embraces seven very dense printed circuitboard types in the heart of the Station, also demands an elaborate real-time operating system and much task-specific software. The Station has internal processing speeds of up to one megapacket per second! A description of the Station is presented in [14]; the first prototype bas been undergoing test and refinement during the final quarter of 1982.

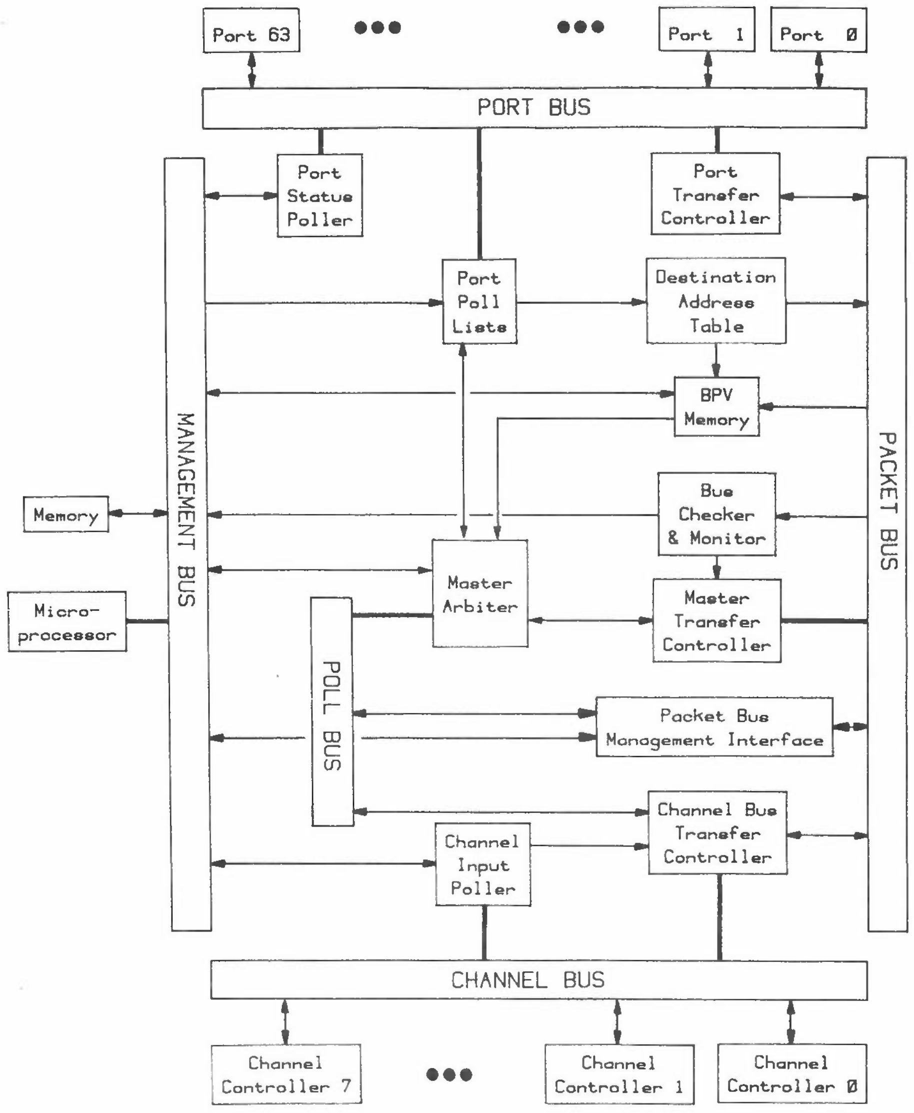


Figure 6. Station Structure

9. WHAT COMES NEXT

Full-performance Stations will go into service at the Polytechnic of Central London and at Bologna University during 1983. It will be possible to configure meaningful interconnections (involving less than four Stations) before final resolution of design details of the high-speed Exchange.

In parallel with the on-going programme of research that will result in many of the refinements mentioned above (e.g. Cambridge Ring interface, IEEE 488 Port, a Speech Port, etc.), the commercial viability of a MININET will be investigated. Operational experience with in-house laboratory users will, of course, be the first proof of the pudding and provide invaluable guidance on future improvements.

10. ACKNOWLEDGEMENTS

We would like to thank the U.K. Science and Engineering Research Council, the Italian Consiglio Nazionale della Ricerche, NATO, EEC, and the British Council for financial support for Project MININET.

11. REFERENCES

[1] Institute of Electrical and Electronic Engineers, "A digital interface for programmable instrumentation", IEEE Standard 488-1978, New York, 1978.

[2] International Electrotechnical Organisation, "An interface system for programmable measuring instruments", IEC Standard 625, Geneva, 1979.

[3] International Standards Organization, "Information processing systems - open systems interconnection - basic reference model", ISO/DIS 7498, April 1982.

[4] Tanenbaum, A.S., *Computer Networks*, Englewood Cliffs, N.J.: Prentice-Hall, 1981.

[5] Martin, J., *Computer Networks and Distributed Processing*, Englewood Cliffs, N.J.: Prentice-Hall, 1981.

[6] Metcalfe, R.M. and D. Boggs, "Ethernet: distributed packet switching for local area networks", *Comm ACM*, vol. 19, no. 7, pp. 395-404, July 1976.

[7] Wilkes, M.V. and D.J. Wheeler, "The Cambridge digital communications ring", Proc. Local Area Communications Network Symposium, Mitre Corp. and NBS, Boston, May 1979.

[8] Morling, R.C.S., "DIM: a network compatible intermediate interface standard", to be published in *Interfaces in Computing.*

[9] Morling, R.C.S. and G.D. Cain, "MININET: a packet-switching minicomputer network for real-time instrumentation", Proc. A.I.M. International Meeting on Minicomputers and Data Communications, Liege, Belgium, January 1975.

[10] Neri, G., R.C.S. Morling and G.D. Cain, "A reliable control protocol for high-speed packet transmission", *IEEE Trans. on Comm.*, vol COM-25, no.

10, pp. 1203-1210, October 1977.

[11] Morling, R.C.S. et al., "The MININET inter-node control protocol", Proc. Symposium on Computer Network Protocols, Liege, Belgium, pp. B4-1/B4-6, February 1978.

[12] Neri, G. et al., "A local area network for real-time instrumentation applications: MININET", PCL Technical Memorandum MN-9, September 1980.

[13] Morling, R.C.S. and G.D. Cain, "A routing protocol that maintains packet sequency", to be presented at Melecon'83, Athens, May 1983.

[14] Morling, R.C.S., "The MININET Station architecture", PCL Technical Memorandum MN-10, January 1983.

MININET

INSTRUMENTATION LAN FOR:

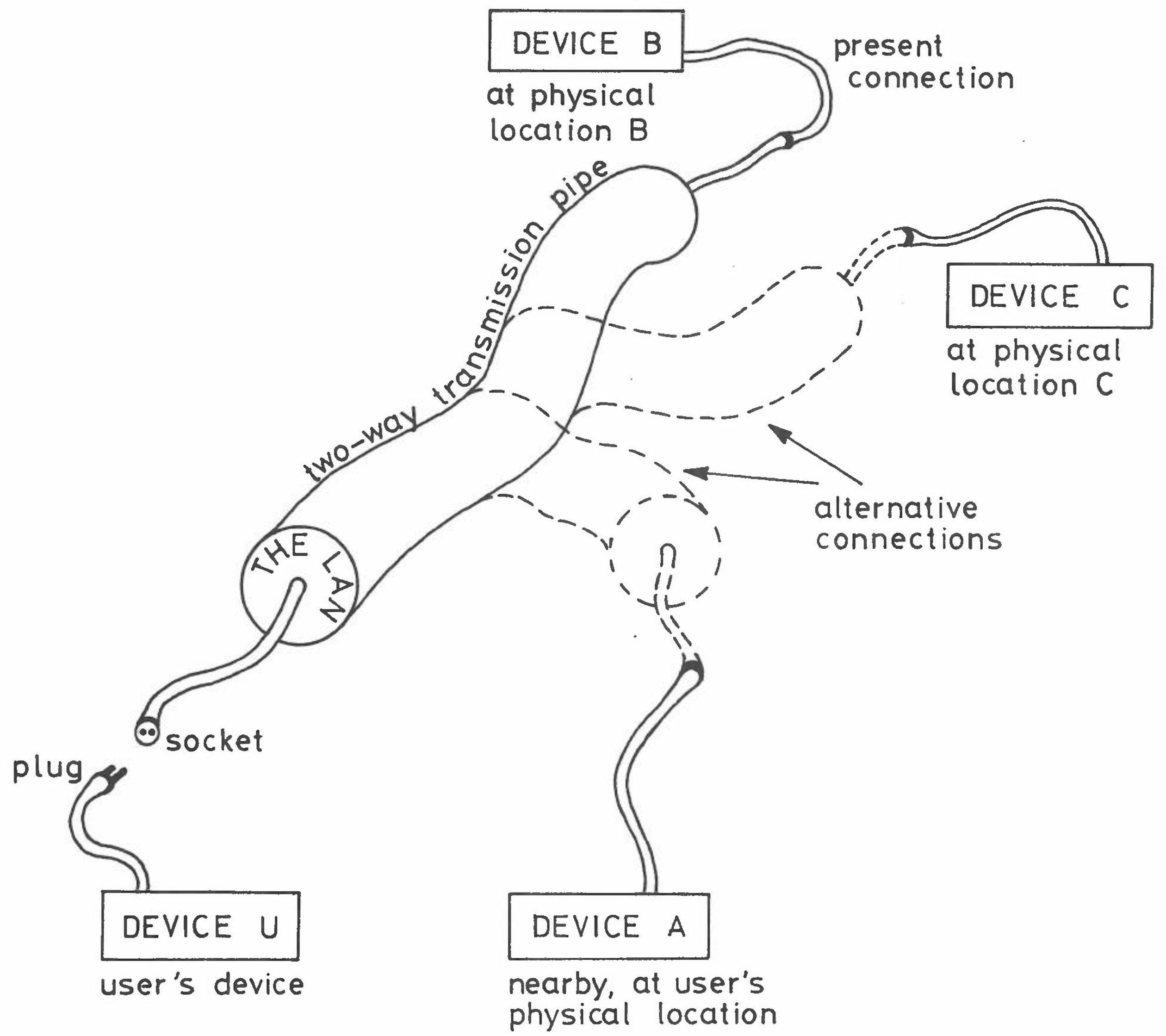
LABORATORY AUTOMATION

PROCESS CONTROL

MANUFACTURING AND ROBOTICS

CONSTRAINTS OF ENVIRONMENT

* REAL REAL-TIME DEADLINES
* MIXED BAG OF USER DEVICES, IN THEIR HUNDREDS
* FLEXIBILITY AND EASE OF USE
* REASONABLY LOW COST
* RELIABLE COMMUNICATION
* RECONFIGURABILITY


DEVICE B
at physical location B
present connection
two-way transmission pipe
DEVICE C
at physical location C
alternative connections
THE LAN
socket
plug
DEVICE U
user's device
DEVICE A
nearby, at user's physical location

WANs

OFFICE AUTOMATION LANs

INSTRUMENTATION LANs

DIFFERENCES AND SIMILARITIES

IF NETWORK LAYER IS NULL

* CAN SOLDIER ON
* LOSE FLEXIBILITY
* LOSE RELIABILITY
* BECOMES LINK TECHNOLOGY DEPENDENT

* INTERCONNECT HETEROGENEOUS USER DEVICE POPULATION

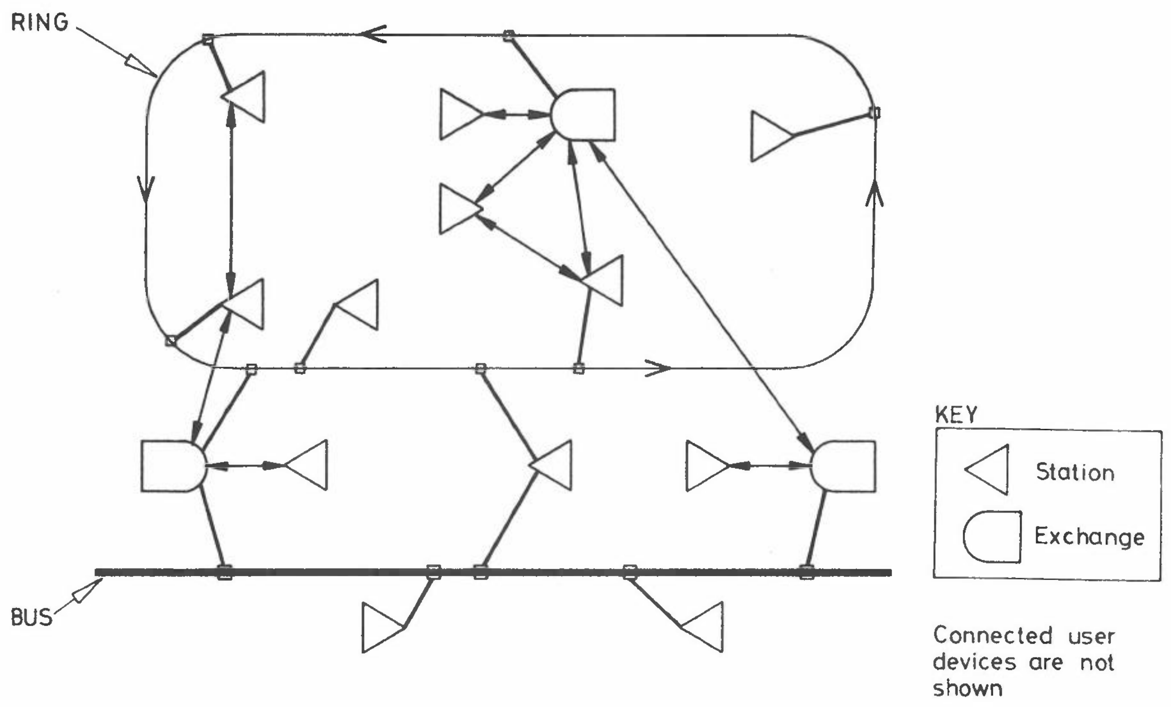
  - TRANSDUCERS
  - A/D, D/A
  - INSTRUMENTS
  - HOMEGROWN SPECIALS
  - COMPUTERS

* CATER FOR NON-INTELLIGENT USER DEVICES

  - INTELL. IS OFTEN OVERKILL
  - NETWORK CONSCIOUSNESS BAD EVEN IF INTELLIGENT (COST/TAP; OPERATION)
  - ANYTHING DIGITAL WELCOME

* CHOICE OF ARBITRARY INTERCONNECTION TOPOLOGY

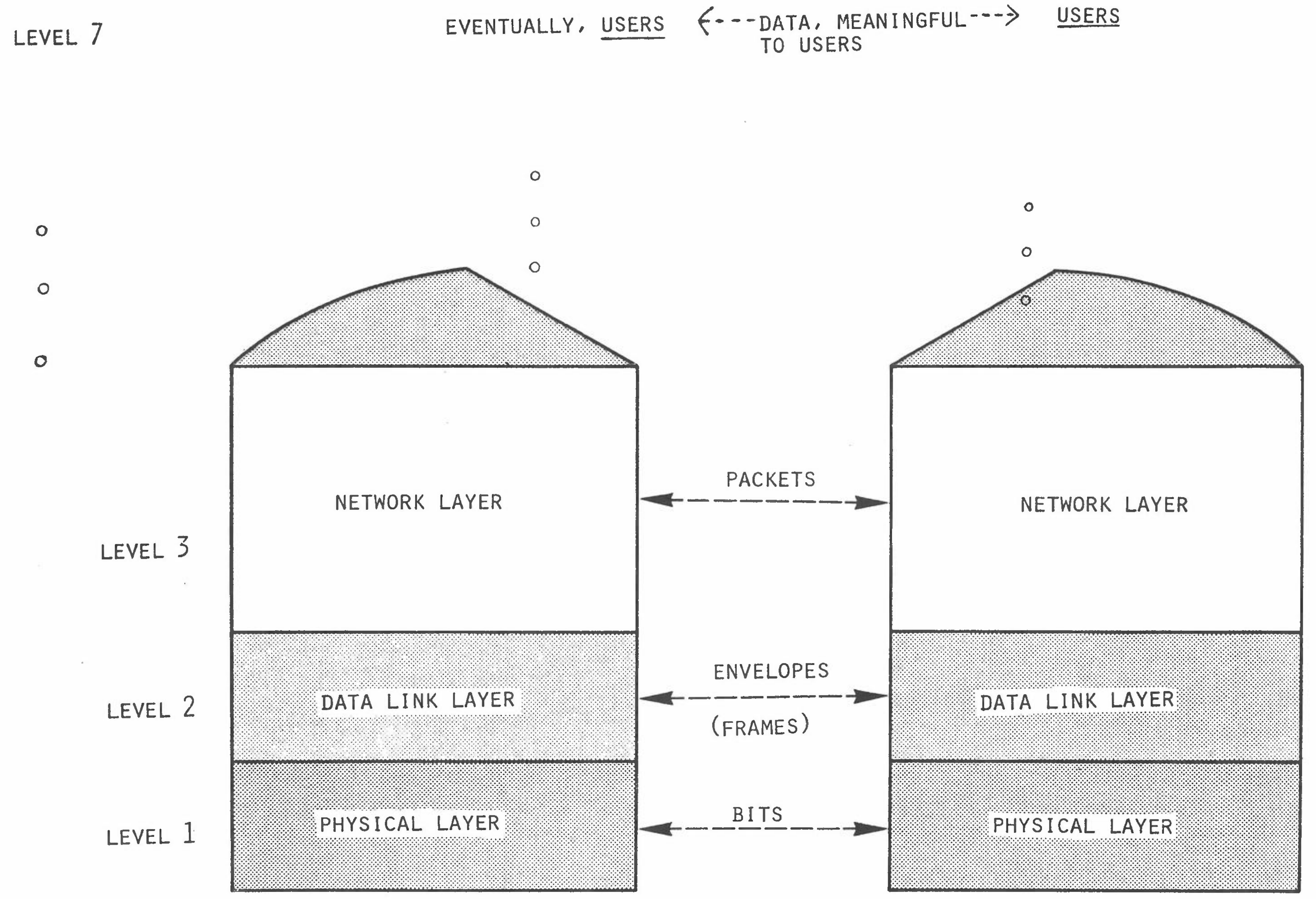
  - SAFETY IN DISTRIBUTION
  - SNAP TOGETHER TO SUIT
  - CAN INCLUDE RINGS AND BUSES


RING
BUS
KEY
Station
Exchange
Connected user devices are not shown

MININET

SERVICES

BUILT ON TOP, AT LEVEL 3

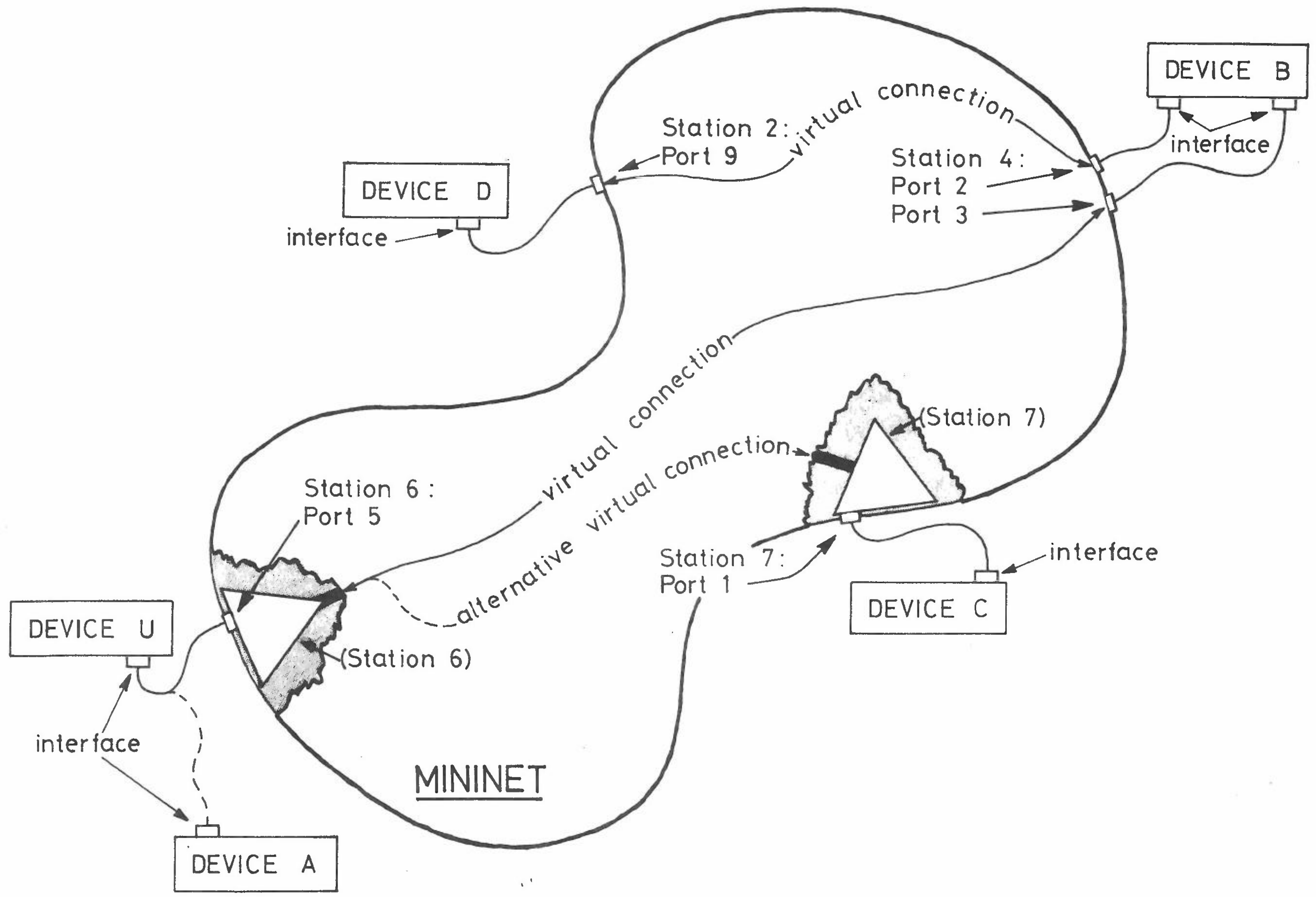

LEVEL 7
EVENTUALLY, USERS
<---DATA, MEANINGFUL--->
TO USERS
USERS
NETWORK LAYER
PACKETS
NETWORK LAYER
LEVEL 3
DATA LINK LAYER
ENVELOPES
(FRAMES)
DATA LINK LAYER
LEVEL 2
PHYSICAL LAYER
BITS
PHYSICAL LAYER
LEVEL 1

* MULTI-MEDIA LINKS

- SLOW/FAST COMBINATIONS
- EVERY LINK INDEPENDENT
- NOT LIMITED BY SLOWEST
- EXAMPLES: OPTICAL FIBRE
  COAXIAL CABLE
  MODEM LINK

* EASILY-REDIRECTED RESOURCE SHARING

  - ONE-TO-ONE VIRTUAL CONNECTIONS
  - "DIAL-UP" CHANGES


DEVICE B
interface
Station 2:
Port 9
virtual connection
Station 4:
Port 2
Port 3
DEVICE D
interface
virtual connection
(Station 7)
alternative virtual connection
Station 6 :
Port 5
Station 7:
Port 1
interface
DEVICE C
DEVICE U
(Station 6)
interface
MININET
DEVICE A

* SUPPORTS VERY HIGH-SPEED COMMUNICATIONS

  - 100 K PACKETS/S
  - 1.6 M BIT/S TO USER (TRANSACTIONS!)
  - 5Ø MICROSECOND HOP DELAY

* ULTRA-HIGH TRANSPARENCY

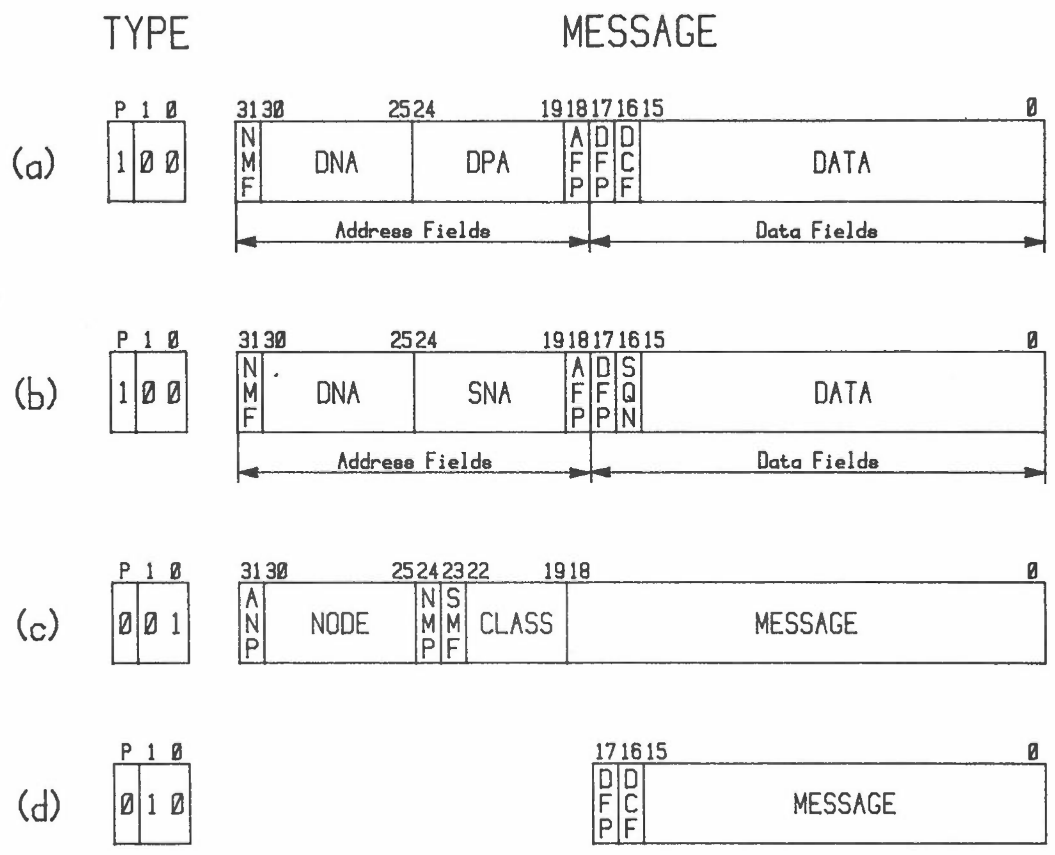
  - DUMB DEVICE DEMANDS IT
  - SMART DEVICE TOO BUSY
  - SHIELDED FROM NETWORK EVOLUTION
  - EVERYTHING SEEMS LOCAL

* SHORT MESSAGES SENT EXPRESS BY WORD SWITCHING

- BURSTY TRAFFIC
- 32-BIT PACKET
- CARRIES 16 USER BITS
- DYNAMIC PHYSICAL PATHWAY CHANGE ● ● ●

WORD SWITCHING ...

- SHORT RESPONSE TIMES
- EFFICIENT BANDWIDTH USE
- NO MESSAGE (DIS)ASSEMBLY
- D.I.Y. PROTOCOLS


TYPE
MESSAGE
(a)
P 1 Ø
1 Ø Ø
31 30 25 24 19 18 17 16 15 Ø
NMF
DNA
DPA
AFP
DFP
DCF
DATA
Address Fields
Data Fields
(b)
P 1 Ø
1 Ø Ø
31 30 25 24 19 18 17 16 15 Ø
NMF
DNA
SNA
AFP
DFP
SQN
DATA
Address Fields
Data Fields
(c)
P 1 Ø
Ø Ø 1
31 30 25 24 23 22 19 18 Ø
ANP
NODE
NMP
SMF
CLASS
MESSAGE
(d)
P 1 Ø
Ø 1 Ø
17 16 15 Ø
DFP
DCF
MESSAGE

* FLEXIBLE YET RELIABLE

- PLANS FOR THE INEVITABLE
- GRACEFUL DEGRADATION
- FAIRNESS CRITERION

HOW DOES A USER CONNECT?

- VIA A DIM INTERFACE
- VIA IEEE 488
- VIA A SPEECH PORT

THE MANUAL OR HOST DIAL-UP OF VIRTUAL CONNECTION

DIM INTERMEDIATE INTERFACE

* DEVICE-TO-DEVICE

* COMPUTER-TO—DEVICE

* LOCAL CONNECTION OR THROUGH MININET

> 16-BIT BIDIRECTIONAL

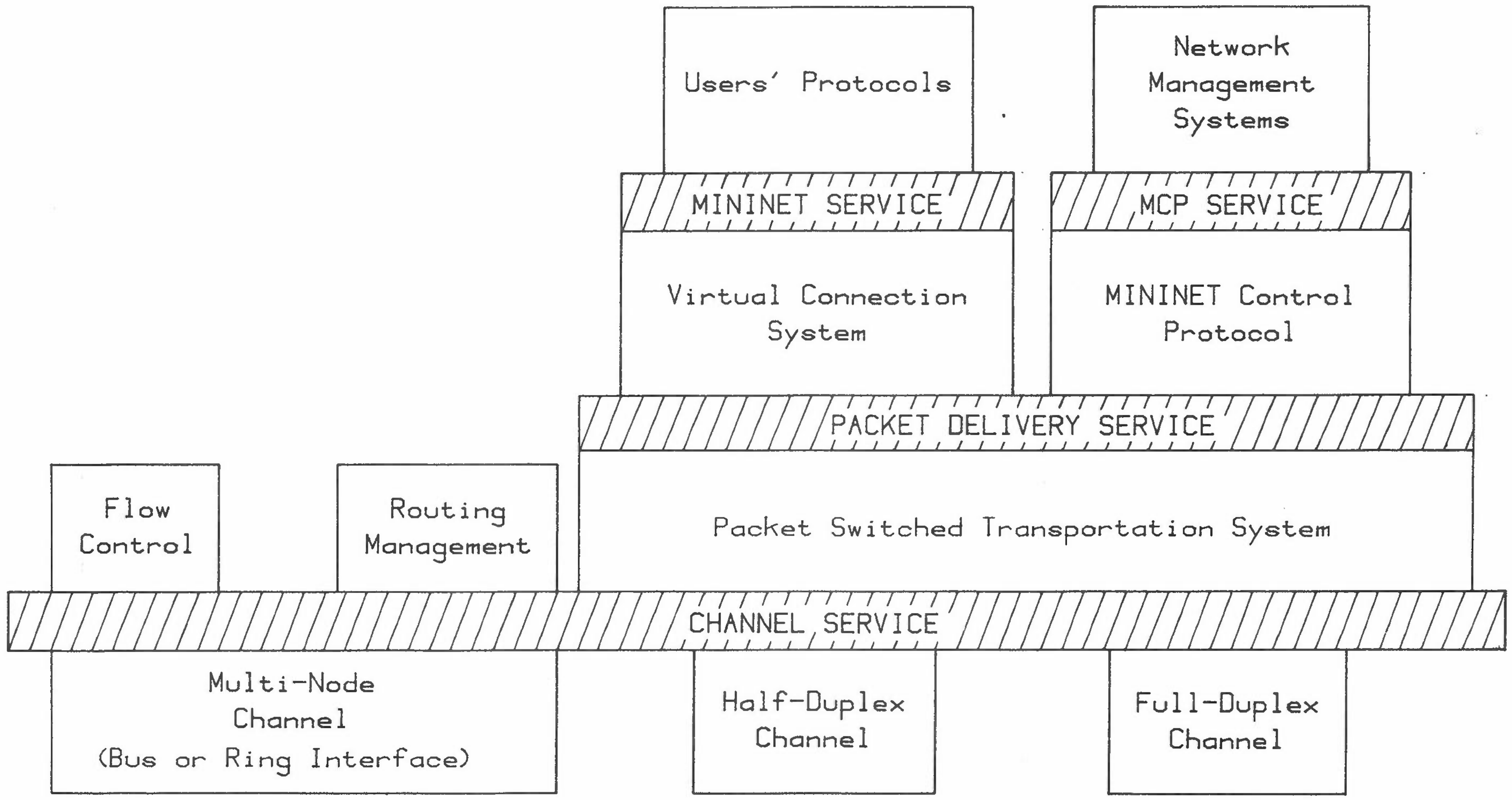
> RATE-CONTROL HANDSHAKE

> CONTROL AND DATA HANDSHAKE

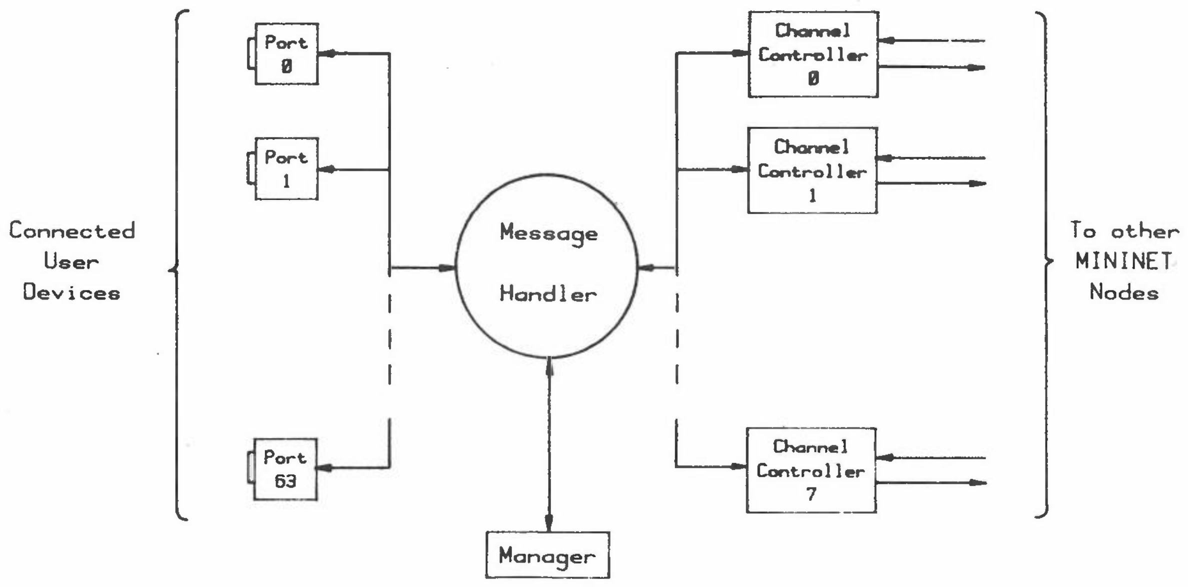
> INTERFERENCE SUPPRESSION - VALIDATION TIMEOUTS

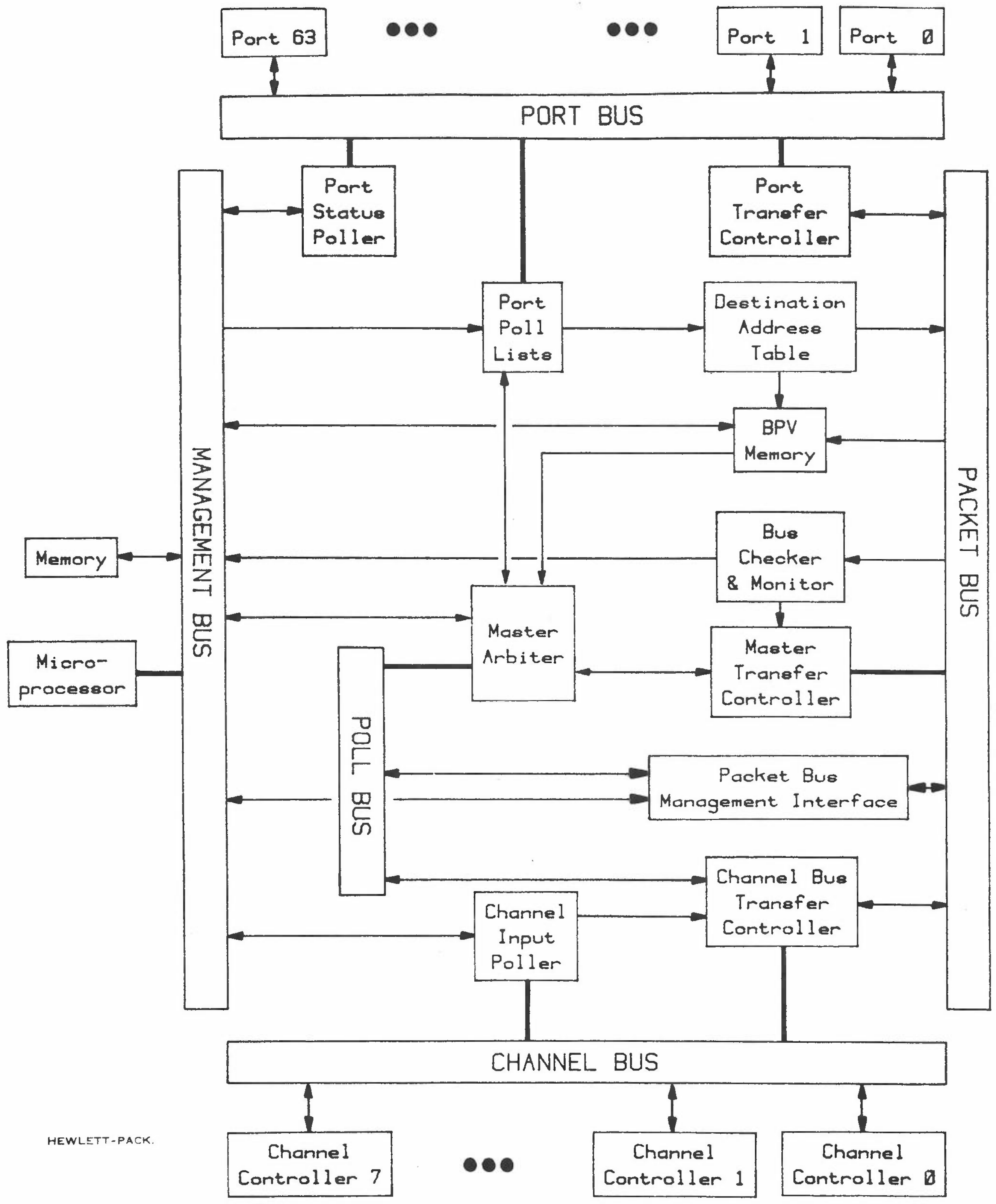
SEQUENTIALITY'S PROFOUND EFFECTS

* LINKS (MLP)
* NETWORK MANAGEMENT (MCP)
* FLOW CONTROL
* REROUTING



MININET HIERARCHICAL STRUCTURE


Connected User Devices
Port 0
Port 1
Port 63
Message Handler
Manager
Channel Controller 0
Channel Controller 1
Channel Controller 7
To other MININET Nodes



HEWLETT-PACK.

THE STRUCTURING OF DISTRIBUTED COMPUTING SYSTEMS

Brian Randell

Computing Laboratory,
University of Newcastle upon Tyne

ABSTRACT

Two recursive structuring principles which aid the construction of sophisticated distributed computing systems are described, namely that:

(i) a distributed computing system should be functionally equivalent to the individual computing systems of which it is composed, even with respect to exception reporting (i.e. the information that a system provides to its environment when it is unable, or perhaps even not designed, to carry out a requested operation), and

(ii) fault tolerant systems should be constructed from generalised fault tolerant components. (Such components try to tolerate, wherever appropriate, their own faults and those reported to them by underlying components, and also being wrongly invoked - both by the components they interact with and the component of which they themselves form part.)

The first principle motivates the use of a strict context-relative naming scheme for all objects in the computing system - taken together the principles enable the problems of constructing coherent systems from heterogeneous components, of incorporating fault tolerance and of providing multilevel security, to be greatly simplified by being treated as essentially separable logical problems.

An operational distributed computing system based on UNIX* and designed in accordance with these principles is used for illustration. This system has been implemented by adding a software subsystem, known as the Newcastle Connection, to each of a set of UNIX systems, so as to construct a distributed system which is functionally equivalent at both the user and the program level to a conventional uniprocessor UNIX system. Prototype extensions of the system,

*UNIX is a Trademark of Bell Laboratories.

January 5, 1983

providing multi-level security and hardware fault tolerance, have also been produced, and are briefly described, as are plans to incorporate non-UNIX systems into the overall distributed system.

1. INTRODUCTION

Careful structuring is of course crucial to the success of any complex computing system design. Appropriately chosen internal interfaces and system components can make the overall system much easier to comprehend, and hence to construct, validate and, if necessary, to modify. This applies whether one is concerned with the design of a large software system, resident on a single computer, or the software and hardware making up a distributed computing system.

In this latter case, allocation of separate software functions to separate computers can make the overall structure of the system much more explicit than when all the software is held in the memory of, and executed by, a single computer. Thus some designers of distributed computing systems have taken as their main structuring principle the identification of functions that can be so treated and their implementation as so-called "servers" - name servers, file servers, boot servers, mail servers, compiler servers, server servers, etc. However, as we will seek to show, there is much more to the topic of structuring distributed computing systems than this. Indeed such an approach, unless carried out in accordance with an appropriate overall system architecture, can lead to systems which are difficult to modify or extend, for example in response to changed workloads.

In this paper various structuring principles and techniques, some but not all of them reasonably well known, are discussed not just in abstract terms but rather as they relate to a distributed system that we have designed and implemented at Newcastle. Our work has in fact involved the development of a software sub-system (called the Newcastle Connection) which can be added to each of a set of physically connected UNIX or UNIX-lookalike systems in order to turn them into a distributed system (which for the purposes of this paper will be called a UNIX United system). Such a system has been operational at Newcastle for some months on a set of PDP11s connected by a Cambridge Ring; pre-release versions of the Connection subsystem have been made available to a small number of other organisations for experiments using various computers and versions of UNIX. Full details of UNIX United and the Newcastle Connection can be found in Brownbridge et al[1], which also surveys a number of related systems developed elsewhere.

The discussion of structuring techniques forms the subject of the next two sections of this paper, with Section 3 also including an overview of the structure of the UNIX United system. These sections provide a logical framework which allows the topics of distributedness, heterogeneity, fault tolerance and security to be treated as essentially separate design issues - in fact as the subjects of Sections 4 to 7, respectively, with Section 8 containing brief concluding remarks.

2. BASIC STRUCTURING TECHNIQUES

One common form of system structuring is that of division of software and/or hardware into (or equivalently, construction of a system out of) a set of co-existing interacting components. Such structuring only makes sense if the interfaces between components are, so to speak, "narrow". Such interfaces are ones which

(i) can be specified more simply than the internal construction of the components that they separate can be described (e.g. procedures with a small number of reasonably simple parameters, or hardware components with relatively few connecting wires), and

(ii) are placed across low bandwidth information transmission paths.

This form of structuring is more easily described than applied, since the correct choice of interfaces (presuming it is not decreed by other factors, such as the requirement to use given pre-existing components) often requires considerable experience and insight. For example, distributed systems that have been structured into multiple specialised servers, but with badly positioned interfaces, could be unnecessarily complicated and/or have very poor performance characteristics indeed, with much time lost moving large amounts of information from computer to computer, or waiting in line for particular heavily congested servers.

The technique of structuring a system by dividing it into a set of co-existing interacting components can be contrasted with the technique which is associated with the phrase "level of abstraction", in which one or more components are constructed using ("on top of") other components. This second form of structuring is commonly thought of, at least in software, in linguistic terms, with an interface defining a language of operations, data types, etc., which the lower level implements and the upper level is programmed in. Perhaps the most notable early application of this program structuring technique to the construction of an operating system was Dijkstra's THE multiprogramming system[2]. In this system a set of conceptually separate levels of abstraction dealt with a set of separate issues, such as processor scheduling and memory allocation. Although this system was a multiprogramming system, the structuring concepts it embodies are as relevant to distributed systems as to centralised systems, and are used extensively in UNIX United.

3. THE PRINCIPLE OF RECURSIVE STRUCTURING

The single most important structuring principle that is reflected in the design of UNIX United is that:

> A distributed computing system should be functionally equivalent to the individual computing systems of which it is composed, even with respect to exception reporting (i.e. the information that a system provides to its environment when it is unable, or perhaps even not designed, to carry out a requested operation).

This recursive structuring principle has previously been advocated and investigated, both at Newcastle and elsewhere, in connection with computer architectures intended specifically for VLSI implementation[3,4]. The principle aim was to design a processor architecture which need not be affected by changes in level of integration - to get away from the apparently inexorable progression of incompatible microprocessor architectures, with ever bigger word lengths and instruction sets, that have accompanied the evolution of VLSI technology. Instead, with a "recursive" architecture, as increased integration levels make it possible, an ever greater number of component processors are fitted within a single chip, so improving its performance without affecting its functionality.

To the best of our knowledge, UNIX United is among the first examples, if it is not the first example, of the application of this principle to operating systems. It is perhaps worth remarking that in computing systems design an architectural concept of any real merit should at least in principle be equally applicable at any of a number of levels, e.g. microprogram, program, operating system, data base access, etc.

Adherence in the design of a distributed system to the recursive structuring principle provides, at the level of complete computing systems, the sort of benefits that the concept of block structure in Algol 60 provided to programmers over what had been available hitherto, in languages such as FORTRAN and COBOL. For example, it makes possible an extremely simple means of joining existing distributed systems together - something that would not be at all easy with, say, the Cambridge Model Distributed System[5], because of the sort of naming and addressing mechanisms it incorporates. Equally, and perhaps less obviously, adherence to the principle facilitates the partitioning of a system into separate pieces, a subject to which we will return in Section 7 in connection with the problem of providing multi-level security.

In essence, the value of the principle is that - by definition - a distributed system which is recursively structured in this way is indefinitely extensible, at least in theory. Indeed UNIX United has been designed with the intention of constructing a very large distributed system, involving both wide and local area networks.

The component systems of which a recursively structured distributed system is constructed must, on the other hand, possess characteristics that are appropriate for the distributed system as a whole - firstly, they must provide (at least the appearance of) parallel processing facilities. This even a uni-processor UNIX does, because of its ability to allow users and their programs to initiate asynchronous processes. In a UNIX United system such processes may, without having to be changed in any way, in fact be run on separate processors, so that quasi-parallelism is transformed into actual parallelism.

Secondly, the various objects within a system (computers, data items, I/O devices, etc.) must be accessible by means which are independent of whether the system is in fact a complete one, or merely a component of a larger system. Thus component computers should support a general "contextual naming" scheme for their various objects. In other words, there should be means for introducing and entering (and leaving) new naming contexts, and all names must be context-relative. This is a characteristic that UNIX possesses by virtue of its hierarchical scheme for naming files, devices and commands, in which directories serve as the required contexts.

```
       root `/' -->       .
                         / \
                        /   \
                       /     \
                    user     lib
                    / \      / \
                   /   \
current           /     \
working  -->   brian   fred
directory       / \     / \
               /   \   /   \
             dir1     b     c
             / \
            /   \
           a     b
```

Figure 1: The UNIX Name Space

Figure 1 shows part of a typical UNIX naming hierarchy. Files, directories, etc., can only be named relative to either the directory which is designated as being the "current working directory" or that which is designated as the "root directory". Thus "/user/brian/dir1/a" and "dir1/a" identify the same file, the convention being that a name starting with "/" is relative to the root directory. Objects outside a context can be named relative to that context using the ".." convention to indicate a parent directory. (Note that this avoids having to know the name by which the context is known in its surrounding context.) The names "/user/fred/b" and "../fred/b" therefore identify the same file, the second form being a name given implicitly relative to the current working directory rather than the root directory.

As its name implies, the current working directory can be changed. In fact the root directory can also be re-positioned. In both cases however, this can be done only by specifying a context-relative name. There is on the other hand no means of specifying an absolute name, relative to the base of the tree, say. The base directory, which is usually but not necessarily chosen as the root directory, can itself be recognised only by the convention that it is its own parent. Moreover all other means provided for identifying any of the various kinds of objects that UNIX deals with, e.g. users, processes, open files, etc., are related back to its contextual naming scheme.

This simple and elegant scheme of context-relative naming has been taken advantage of in UNIX United by identifying individual component UNIX systems with directories in a larger name space, covering the UNIX United system as a whole. Any directory can be associated with a separate UNIX machine - in Figure 2 we show how a UNIX United system spanning an entire university might be created from the machines in various university departments, using a naming structure which matches the departmental structure (without regard to the actual topology of the underlying communications networks).

```
                              .
                             / \
                            /   \
                           /     \
                          /       \
                         /         \
                       CS          EE     Maths  ....
                       / \         /¦\
                      /   \       / ¦ \
                    U1     \     /  ¦  \
                   /¦\     U2  U1 U2  U3
                  / ¦      /\  /\ /\  /\
                          /   /  /  \   \
                             /       \
```

Figure 2: A University-Wide UNIX United System

The figure implies that from within the Computing Science Department's U1 machine, files on its U2 machine will normally have names starting "/../U2" and files on the machine that the Electrical Engineering Department has also chosen to call "U2" will need to be identified with names starting "/../../EE/U2". (UNIX has various means for, in effect, abbreviating lengthy names, which need not concern us here.)

By taking advantage of UNIX's contextual naming scheme in this way, it has been possible to produce a distributed system which is functionally equivalent at both the user and the program level to a conventional uni-processor UNIX system, even with respect to error reports. All the standard UNIX conventions, e.g. for protecting, naming and accessing files and devices, for input/output redirection, for inter-process communication, etc., are applicable without apparent change to the system as a whole. All issues of inter-processor communication, networking protocols, etc., are hidden. Thus the standard UNIX facilities can be used without concern for the fact that several machines may be involved, and the user need have no knowledge of what data and messages flow when, or between which machines, or which processor actually executes any particular program.

In fact the present implementation of UNIX United runs programs on the machine within whose file store their program code is held. Thus the command line

```
/../U1/sort /../U2/data ¦ /../U3/summarise
```

which might have been entered on a terminal logged in to U4, say, causes the `sort' program to be run on machine U1, with its data being fetched from U2 and the results being piped to the `summarise' program running on U3 in parallel with `sort'. On the other hand if U1 and U3 were merely directories on a single UNIX machine, `sort' and `summarise' would have run in an interleaved fashion.

Adoption of the recursive structuring principle thus has a profound (and highly beneficial) effect on the usability of a distributed system. It also provides, in conjunction with the principle of the separation of logical concerns, a number of valuable guidelines as to how to tackle the various implementation issues, including the provision of fault tolerance and multi-level security, and also the construction of a coherent system from a collection of heterogeneous components.

4. DISTRIBUTEDNESS

In our view, the structuring principles discussed above make it particularly natural to regard "distributedness" (i.e. the fact that a system incorporates a set of autonomous yet interacting computers) as providing a design problem which is clearly separable from various other issues, most notably fault tolerance, with which many papers and designs often link it apparently inextricably, e.g. Popek[6]. We regard the problem as involving two principal issues, namely those of:

(i) routing each request for activity to the appropriate component computing system, from the one in which it originated, and

(ii) preserving the appearance of a single overall recursively structured name space, and hiding any local addressing mechanisms used inside the component computing systems in support of this name space.

The provision of atomic actions, for example, is regarded as a separate issue (addressed in Section 6), since such facilities are at least in principle of as much relevance to a computing system which merely provides the appearance of parallel (and hence possibly interfering) processes using a single processor as one in which there is actual parallelism. In other words, atomic actions are as relevant to a time sharing system as to a distributed system.

We have accordingly tried to identify the minimum set of facilities that are needed for the provision of distributedness (in a recursively structured system) and to implement them in a clearly separate mechanism. UNIX United has in fact been implemented merely by adding a subsystem, in the form of a software layer, to an otherwise unchanged UNIX system. In direct analogy with the THE hierarchy of levels of abstraction, other equally strictly distinguished software layers, relevant to other separable logical concerns, can be placed above (or below) the Newcastle Connection layer, as appropriate.

The positioning of the Connection layer is governed by the structure of UNIX itself. In UNIX all user processes and many operating system facilities (such as the `shell' command language interpreter) are run as separate time-shared processes, able to interact with each other, and the outside world, only by means of `system calls' - effectively procedure calls on the resident nucleus of the operating system, the UNIX kernel. The Connection is a transparent layer that is inserted between the kernel and the processes. It is transparent in the sense that from above it is functionally indistinguishable from the kernel and from below it appears to be a set of normal user processes. It filters out system calls that have to be re-directed to another UNIX system (for example, because they concern files or devices on that system), and accepts calls that have been directed to it from other systems. Thus processes on different UNIX machines can interact in exactly the same way as processes on a single machine. (There is a quite separate issue of whether UNIX needs additional forms of inter-process communication, e.g. for synchronous message passing between unrelated processes. Clearly, if such facilities were added to the UNIX kernel the Connection layer would have to be extended in order to ensure that they worked for processes that happened to be on different machines.)

Since system calls act like procedure calls, communication between the Connection layers on the various systems is based on the use of a remote procedure call protocol[7], and is shown schematically below.

```
 --------------------                          --------------------
|User programs,      |                        |User programs,      |
|non-resident        |                        |non-resident        |
|UNIX software       |                        |UNIX software       |
|--------------------|    remote procedure    |--------------------|
|Newcastle Connection|<---------------------->|Newcastle Connection|
|--------------------|         calls          |--------------------|
|UNIX Kernel         |                        |UNIX Kernel         |
 --------------------                          --------------------

         UNIX1                                          UNIX2
```

Figure 3: The Position of the Connection Layer

A slightly more detailed picture of the structure of the system would reveal that communications actually occur at the hardware level, and that the kernel includes means for handling low level communications protocols. However all such issues are hidden from the user of UNIX United.

It is of course still left to each UNIX programmer to choose to implement a given algorithm in the form of a single process, or alternatively as a set of interacting processes, so as to take advantage of the quasi-parallelism in UNIX, and perhaps real parallelism in UNIX United. Thus the existence of the Newcastle Connection still leaves open the question of whether a centralised or (logically) distributed implementation of, say, a data base manager is most appropriate in given circumstances - physical distribution has however been subsumed within logical distribution.

4.1. Names and Addresses

In a recursively structured system each component computer possesses what appears to be a complete name space, but which in fact is just part of the overall name space. Thus one of the consequences of distributedness is the requirement for some means of combining these component name spaces.

The technique that we have evolved for this purpose in UNIX United is as follows. Each component UNIX system stores just a part of the overall naming structure. In fact each system stores the representation of its own section of the naming tree. However each system also stores a copy of those parts of the overall naming structure that relate it to those other UNIX systems with which it is directly connected in naming terms (i.e. which can be reached via a traversal of the naming tree without passing through a node representing another UNIX system).

```
        (base) .
              / \
             /   \
            /     \
           A       B
          / \     / \
         /   \   /   \
        E     F D     C
                     / \
                    /   \
                   G     H
```

Figure 4: Representing the Name Space

In Figure 4, if "directories" A, B and C are associated with separate UNIX systems, the parts of the tree representation stored in each system are as follows:

UNIX-A: A,B,E,F,(base)

UNIX-B: A,B,C,D,(base)

UNIX-C: B,C,G,H

It is assumed that shared parts of the naming tree are agreed to by the administrators of each of the various systems involved, and do not require frequent modification - a major modification of the UNIX United naming structure can be as disruptive as a major modification of the naming structure inside a single UNIX system since names stored in files or incorporated in programs (or even just known to users) may be invalidated.

It is not reasonable to assume that all accessing of all types of object in a distributed system will be performed using general contextual names each and every time. Rather, in the interests of efficiency, and in order to exploit actual hardware provisions, names will when appropriate be bound to what are effectively addresses, of limited scope and validity, so that repeated accesses to a given object can then be made by just using its address.

For example, in UNIX the act of opening a file (identified by a contextual name) causes what is effectively an address, local to the current process, to be assigned for use in subsequent accesses to that file. This address (a UNIX "file descriptor") will be valid only until the file is closed. If the file is to be accessed by a process on a remote system, a file descriptor which is valid on that system must be used. However this file descriptor must be associated with the relevant file descriptor on the system holding the file. In UNIX United this task is carried out within the Connection layer, using appropriate mapping tables. (There is of course more to the subject of names and addresses than this - see for example Saltzer[8]. Indeed the facilities in UNIX for name binding have some inadequacies. However the job of the Connection layer is to produce a fully compatible distributed version of UNIX, and not to "improve" its functionality in any way.)

4.2. And As For Name Servers ..

Given the comments in the Introduction about structuring a distributed system out of specialised servers on separate computers it is perhaps worth describing explicitly how one such server concept, that of a "name server", fits into the UNIX United scheme. The basic function of a name server is to provide a central repository for information regarding the physical addresses of the various other components of the distributed system, information that can then be used to enable these components to be accessed directly.

A UNIX United system can easily be set up, using a Ring network, say, to work in just this way, as is illustrated in Figure 5.

```
              NS                             NS
              |                             /  \
         -------------                     /    \
        |             |                   /      \
     U5-|             |-U1               /        \
        |             |                 /          \
     U4-|             |-U2             /            \
        |             |               /              \
         -------------              U1  U2   ...      U5
              |                     /\  /\            /\
              U3
```

Figure 5: A Name Server

Here all but one of the component UNIX systems are made subservient, in terms of the global naming tree, to the one other system, labelled NS in the diagrams. This system will contain hardware addresses (ring port numbers) for all of the others, each of which will hold the hardware address of just the NS system. If U1, say, needs to access a file on U2 it can `open' the file using a name which starts "/../U2" - hence the name of this section! The `open' system call will have to access NS in order to check permissions, but will in due course return the port number of U2 so that thereafter reads and writes to the file will go directly to U2, and not involve NS.

The name server idea is thus seen as just a specialised usage of the general UNIX United scheme, and in particular of the route optimisation that the Connection implements amongst a set of UNIX systems that are all in the same hardware address space, be it a Ring, an Ethernet, an X25 network, or whatever.

5. **HETEROGENEITY**

Much work on distributed systems, particularly on network protocols, mixes up two separate issues, namely distributedness and heterogeneity. Thus typical file transfer protocols, virtual terminal protocols, etc., are as much concerned with incompatibilities between computing systems as the fact that several computing systems are involved. Our approach, based on isolating and then separately solving the problem of distributedness, would appear to be of relevance solely to computing systems composed of identical component computing systems. This is not the case - the components just have to be functionally equivalent to each other, a task that is made easier by the fact that there are defined ways of responding to system calls which cannot be carried out, for whatever reason.

For example, one could connect together UNIX systems which have little or no file storage with other systems that have a great deal - i.e. construct a UNIX United system out of workstations and file servers. Almost all that is necessary is to set up the naming tree properly.

Moreover since the Connection layer is independent of the internals of the UNIX kernel, it is not even necessary for the Connection layer to have a complete kernel underneath it - all that is needed is a kernel that can respond properly (even if only with exception messages) to the various sorts of system call that will penetrate down through, or are needed to support, the Connection layer. In fact the Connection layer itself can be economised on, if for example it is mounted on a workstation that serves as little more than a screen editor, say, and so has only a very limited variety of interactions with the rest of the UNIX United system. All that is necessary is adherence to the general format of the inter-machine system call protocol used by the Newcastle Connection, even if most types of call are responded to only by exception reports. (In fact when a UNIX United system is implemented using a number of different types of computer, low level incompatibilities such as differing number representations might not be fully hidden from users.)

Thus the syntax and semantics of this protocol assume a considerable significance, since it can be used as the unifying factor in a very general yet extremely simple scheme for putting together sophisticated distributed systems out of a variety of size and type of component - an analogy we like to make is that the protocol operates like the scheme of standard-size dimples that allow a variety of shapes of LEGO children's building blocks to be connected together into a coherent whole. In particular it can be seen as unifying the two apparently distinct forms of structuring discussed in Section 2, since essentially the same interface is used both to co-existing and to underlying components.

In fact one example of the use of specialised components was mentioned earlier, i.e. the name server. However, though the name server was discussed as though it was a standard UNIX system (perhaps even with its own files and processing activities) this does not have to be the case. Rather, if it is functioning solely as a name server it could well make sense for it to have been implemented specially, and not based on a UNIX kernel, so long as it responds properly, if only with exception messages, to the various UNIX system calls.

Another specialised component that is being investigated at Newcastle is a terminal concentrator. The concentrator is designed to serve as part of an existing campus network, serving various sorts of host computing systems, and is in no way related to UNIX. However it is now being extended so as to have a (very limited) remote UNIX system call interface, so that it can be linked to a UNIX United system, from which it will appear to be a conventional UNIX system whose naming tree contains just terminals.

Another development that is being considered is that of providing a limited remote UNIX system call interface on a totally different operating system. Initially just the basic system calls concerned with file accessing would be supported, and mapped into equivalent facilities within this other system. The simplicity and extensibility of this approach contrast favourably with the more conventional one of having each operating system support a general file transfer protocol, particularly since it enables a remote file to be accessed and updated selectively.

Perhaps the most ambitious approach to the problems of linking heterogeneous computing systems together is the Open Systems Interconnection (OSI) scheme[9]. This involves the creation and definition of what are hoped will prove widely acceptable abstractions of a number of concepts that exist in differing forms in different systems, such as files, processes and transactions. In fact the OSI scheme can be viewed as an attempt to specify an abstract operating system, unfortunately one whose adequacy and merits cannot be known until its functions have been mapped successfully on to one or more probably uncongenial host operating systems. In comparison our UNIX-based approach provides a form of "coherent heterogeneity" based on a proven set of abstractions, already successfully implemented on a wide and ever-growing variety of different hardware. Moreover it is our belief that the UNIX system calls are sufficiently simple, yet general, to be used as a common basis on to which

LEGO is a Registered Trademark of LEGO Systems A/S.

one should be able to map many of the facilities of a wide variety of different operating systems. Thus in relation to OSI's famous seven levels of protocols, one could say that the present UNIX United hides the bottom four levels and in some senses does away with the need for the other three, substituting for them the full UNIX system call interface. (In fact it would be possible to implement the Newcastle Connection at various other levels of the OSI model.)

6. FAULT TOLERANCE

System structuring techniques play a very large role in the provision of effective fault tolerance, as is discussed at length by Anderson and Lee[10]. This describes our (of course!) recursive approach to the construction of fault tolerant systems out of fault tolerant components, such that, at least in principle, each level of component can contain facilities for trying to tolerate:

(i) faults in underlying components that are reported to it,

(ii) its own faults, and

(iii) faulty invocation of the component by its environment, i.e. the enclosing component, or a co-existing component with which it is interacting.

Compared to a centralized system, a distributed system provides new opportunities for the provision of high reliability by means of fault tolerance, and also new types of fault that could impair reliability unless properly tolerated. These are logically separable issues and should be treated as such. Moreover they are also separable from any opportunities or requirements for fault tolerance that would exist in an equivalent non-distributed system.

In line with this attitude regarding the separation of logical concerns, UNIX United is structured so that the only reliability problems which are treated within the Connection layer are those which arise specifically from the fact that the system is distributed. Nevertheless the internal structure of the Connection layer is itself worthy of note.

The Connection layer uses a remote procedure call (rpc) protocol which addresses the problems caused by breakdowns ("crashes") of the component computers and communication links, and the occasional loss of messages across the links. In these circumstances it could be all too easy for a remote procedure to be accidentally executed several times - our rpc protocol attempts to prevent this and to achieve an "exactly once" semantics[11]. There remain the problems of the computer which is trying to make a (perhaps related) series of remote procedure calls itself crashing on occasion, or of the Connection layer, despite its best efforts, being unable to achieve all the requested calls. These problems are not regarded as the province of the Connection layer. Rather they are essentially similar to problems that can arise in a centralized (multi-programming or time-sharing) system - problems for which recoverable atomic actions (ones which are guaranteed either to succeed or to do nothing) provide an answer. (All of these issues are treated in much greater detail in papers relating to the rpc protocol.)[12,13]

Though it follows from application of the recursive structuring principle, it is worth mentioning explicitly that the Connection layer reports any errors that it cannot recover from in terms similar to those used by the UNIX kernel. Thus it reports merely that a file cannot be opened, rather than, say, that the communications line to the machine containing the file is not operational. (In practice, facilities to aid fault location and repair may well be needed, but this is regarded as a separate issue from that of exception reporting for purposes of fault tolerance.)

Support for atomic actions per se is not regarded as part of UNIX United, since it would augment the functionality of UNIX itself. However an appropriate further software layer is being developed which will provide UNIX, and hence UNIX United, with atomic actions which are recoverable, at least with respect to file usage. It is based on the Distributed Recoverable File System[14] developed earlier at Newcastle for UNIX. In essence the new layer will just provide three additional system calls:

(i) Establish Recovery Point (i.e. start state-saving, and locking files),

(ii) Discard Recovery Point (i.e. discard saved state, and unlock relevant files), and

(iii) Restore Recovery Point (i.e. go back to latest uncommitted recovery point).

Adding this layer to the system is best done by placing it between the kernel and the Connection layer, whose provisions will therefore have to be augmented to deal with the three additional system calls. (In the case of the Discard Recovery Point call, it might well be thought necessary to incorporate a simplified form of "two-phase commit protocol"[15], which would involve the provision of another system call "Prepare to Discard Recovery Point" by the Atomic Action layer. This should minimise the risk of having some but not all the component UNIX systems complete their Discard Recovery Point calls.) The resulting software structure for each component computer will be as shown in Figure 6 below:

```
 -----------------------------
|      |     |     |     |      |
|      |     |     |     |      |
|    User Processes, etc.       |
|      |     |     |     |      |
|      |     |     |     |      |
|-----------------------------|
| Newcastle Connection,        |
| with rpc and two-phase       |
| commit protocols             |
|-----------------------------|
| Atomic Action support        |
|-----------------------------|
| UNIX kernel, with            |
|        message passing       |
 -----------------------------
```

Figure 6 : Fault Tolerance Layers

Another well known form of fault tolerance is that of using replication and majority voting, typically with the aim of masking operational hardware faults. A prototype extension to UNIX United has already been constructed which uses this approach. It has involved adding an additional transparent software sub-system (the Modular Redundancy layer) to each of a number of UNIX systems on top of their Connection layers. Copies of a conventional application program and its files can then be loaded onto each of three systems and run so that file accesses are synchronized and voted upon. Any malfunctioning computer so identified by the voting is automatically switched out and in due course another switched in to replace it.

This of course is not a new idea - a well-known computing system using this technique is the SIFT system[16]. The point is that the technique is very simple to implement when it is separated from issues of distributedness. Needless to say, given that the Modular Redundancy layer is transparent, one can envisage using both it and the Atomic Action layer together, the latter having the task of trying to cope with situations where the problem is not a hardware fault, but one arising, say, from erroneous input data.

The significance of the simplicity, generality and mutual independence of these various mechanisms when used in conjunction with the notion of recursive structuring is considerable. Complexity is one of the major impediments to reliability, so that complicated and needlessly interdependent fault tolerance mechanisms are more likely to reduce than to improve reliability, because of the danger of situations arising, particularly during error recovery and system reconfiguration, that have not been catered for properly.

7. SECURITY

The recursive structuring principle implies that the security provisions existing in component computing systems must be mirrored exactly by the distributed system as a whole. In fact UNIX United allows each constituent UNIX system to have its own named set of users, user groups and user password file, its own system administrator ("super-user"), etc. Each constituent system has the responsibility for authenticating (by user identifier and password) any user who attempts to log in to that system. Any Connection layer, when receiving messages containing system calls diverted to it by the Connection layer on some other system, then only needs to authenticate the identity of the system which is the source of its messages.

From an individual user's point of view, therefore, though he might have needed to negotiate not just with one but with several system administrators for usage rights beforehand, access to the whole UNIX United system is via a single conventional `login'. Subject to the rights given to him by the various system administrators, he will then be governed by, and able to make normal use of, the standard UNIX file protection control mechanisms in his accessing of the entire distributed file system. In particular there is no need for him to log in, or provide passwords, to any of the remote systems that his commands or programs happen to use. This approach therefore preserves the appearance of a totally unified system, without abrogating the rights and responsibilities of individual system administrators.

Though the standard UNIX security facilities, when carefully used, are as good if not better than those of most other time-sharing systems, they are quite inadequate against determined attack by would-be penetrators. There are many situations where the potential costs of a security violation, and the danger of deliberate attempts to subvert the security controls, are so great that positive assurances regarding the security of a system are required. (This is particularly the case if a multilevel security policy is involved.) It then does not suffice merely to have "plugged" all the various gaps that previous penetrations have revealed. Rather, a compelling argument guaranteeing that the system has been designed and implemented so as to adhere to the required security policy must be supplied. In these circumstances it is essential to construct the system in such a way that the mechanisms on which security depends are clearly identified, and simple enough for their adequacy to be manifest - ideally for their correctness, with respect to some formal statement of the required security policy, to be proven formally, perhaps by, or with the aid of, a program verification system.

```
                  ----------   ------
                 |Top Secret| |Secret|
                 |   UNIX   | | UNIX |
                  ----------   ------
                      |           |
                      Z           Z
                      |           |
                  ----------------------
 ------------    |                      |    -----------
|Unclassified|--Z--|                    |--Z--|Secure     |
|    UNIX    |   |                      |   |File Server|
 ------------    |                      |    -----------
                  ----------------------
                    |             |
                    Z             Z
                    |             |
                  ------     ------------
                 |Secret|   |Unclassified|
                 | UNIX |   |    UNIX    |
                  ------     ------------
```

Figure 7: Multi-level Secure UNIX

Two important types of security mechanism are those that prevent information flow, and those that monitor and mediate such information flow as is allowed, between system components that cannot be trusted to adhere to the security policy, perhaps because of their complexity. The recursive structuring technique can readily be used to construct a system that provides multi-level security, by allocating separate (untrusted) general-purpose component computers to different security levels and implementing appropriate (trusted) security mechanisms as transparent additions to the inter-processor communication links. A prototype of such a system has in fact already been constructed (indeed in just a few days) based on UNIX United, using encryption-based security mechanisms (called "Z-boxes") to prevent information flow between security regimes, and to control the types of security reclassification allowed. This system is portrayed in Figure 7 above.

Construction of a much more sophisticated version of this system is now planned, a description of which is given in Rushby and Randell[17]. The system structuring techniques involved enable us to have some confidence that the design of the security-relevant aspects of the system will be simple enough to be amenable to formal specification and verification. There is also every reason to believe that the system will not suffer the sort of severe performance degradation that has resulted when attempts have been made to provide multi-level security in a general purpose operating system running on a single computer. Furthermore, there should be nothing to prevent the immediate incorporation of the sort of fault tolerance mechanisms described earlier, without impacting the trustworthiness of the security mechanisms, due to the careful separation of the various issues involved.

8. CONCLUSIONS

In our view, the ideas that we have tried to encapsulate in the form of two "structuring principles" provide a surprisingly effective and constructive methodology for the design of distributed systems, and for dealing with issues such as reliability and security. Moreover they give a means of making a coherent system out of a heterogeneous set of components - in part by hiding the heterogeneity behind a facade of homogeneity, in part by treating it as exceptional behaviour for which fault tolerance provisions can be made.

Certainly our experience with UNIX United provides what we regard as strong evidence for the merits of this methodology. As reported in[1], a very useful distributed system, enabling full remote file and device access, was constructed within a month of starting implementation of the Connection layer. Needless to say, the fact that - due to the transparency of the Newcastle Connection - it was not necessary to modify or in most cases even understand any existing operating system or user program source code was a great help! In a very few months this system had been extended to cover remote execution, multiple sets of users, etc., two prototype extensions of the system, for multi-level security and hardware fault tolerance, had been successfully demonstrated, and the design of others commenced. However we still feel we have barely begun to explore all the many possible ramifications of the scheme, and of course there are many evaluation exercises and engineering improvements to be investigated.

What has been presented here as a discussion of structuring principles for the design of distributed computing systems could equally well be viewed as a rationale for the design of UNIX United. It would be nice to be able to report that the process of designing UNIX United had been guided, at all times, by explicit recognition of these principles. In practice the above account is as much a rationalisation of, as it is a rationale for, the design of UNIX United. The various structuring ideas, in particular those on fault tolerant components and on recursive architectures, had already been a subject of much work at Newcastle, but the work that led to the specification and detailed design of the Newcastle Connection has contributed to, as well as greatly benefitted from, our understanding of system structuring issues. Equally it owes much to the external form (if not internal design) of UNIX itself - the only operating system we know of which is at all close to being an ideal component of a recursively structured distributed computing system. Nevertheless we would not wish to give the impression that UNIX is perfect, and that these structuring ideas are relevant only to UNIX and UNIX-like systems. Rather, we believe that the ideas provide an interesting perspective from which to judge the merits of existing or planned systems, and can help to guide further work on the design of operating systems and distributed systems.

9. ACKNOWLEDGEMENTS

The discussion of structuring issues presented in this paper owes much to the work of the author's many colleagues, at Newcastle and elsewhere, over a number of years. Recent work specifically on the Newcastle Connection, however, has been mainly in close collaboration with Lindsay Marshall, Dave Brownbridge, Fabio Panzieri and Jay Black. The work on multi-level security has been largely carried out with John Rushby, and that on hardware fault tolerance with Li-Yi Lu. Research on distributed computing systems at Newcastle is sponsored by the U.K. Science and Engineering Research Council and the Royal Signals and Radar Research Establishment.

References

1. D. R. Brownbridge, L. F. Marshall, and B. Randell, "The Newcastle Connection - or UNIXes of the World Unite," Software Practice and Experience Vol. **12**(12) (Dec. 1982).

2. E.W. Dijkstra, "The Structure of the `THE' Multiprogramming System," Communications of the ACM Vol. **11**(5), pp.683-696 (May 1968).

3. P. C. Treleaven and R. P. Hopkins, "A Recursive Architecture for VLSI," pp. 229-238 in Proc. 9th Ann. Symp. on Computer Architecture, IEEE (26-29 April 1982).

4. W. T. Wilner, Recursive Machine, Palo Alto Research Center, Xerox Corp., Palo Alto, Calif. (1980).

5. M. V. Wilkes and R. M. Needham, "The Cambridge Model Distributed System," Operating System Review Vol. **14**(1), pp.21-28, ACM (Jan. 1980).

6. G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel, "LOCUS: A Network Transparent, High Reliability Distributed System," Operating Systems Review Vol. **15**(5), pp.169-177, ACM, (Proc. ACM 8th Conf. Operating Systems Principles, Asilomar, Calif.) (Dec. 1981).

7. F. Panzieri and S. K. Shrivastava, "Reliable Remote Calls for distributed UNIX: An implementation study," pp. 127-133 in Proc. Second Symp. on Reliability in Distributed Software and Database Systems, IEEE, Pittsburg (July 1982).

8. J. H. Saltzer, "Naming and Binding of Objects," in Lecture Notes in Computer Science 60, ed. R. Bayer, R.M. Graham and G. Seegmueller, Springer-Verlag, Berlin (1978).

9. H. Zimmerman, "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection," IEEE Trans. on Communicatons Vol. **COM-28**, pp.425-432 (April 1980).

10. T. Anderson and P.A. Lee, Fault Tolerance: Principles and Practice, Prentice-Hall (1981).

11. B. J. Nelson, Remote Procedure Call, Ph.D. Thesis, Computer Science Dept., Carnegie-Mellon Univ., Pittsburg, Pa. (1981).

12. S. K. Shrivastava, "Structuring Distributed Systems for Reliability and Crash Resistance," IEEE Trans. Software Eng. Vol. **SE-7**(4), pp.436-447 (July 1981).

13. S. K. Shrivastava and F. Panzieri, "The Design of a Reliable Remote Procedure Call Mechanism," IEEE Trans. on Computers (July 1982).

14. M. Jegado, "Recoverability Aspects of a Distributed File System," Software Practice and Experience (To appear).

15. J.N. Gray, "Notes on Data Base Operating Systems," pp. 393-481 in Lecture Notes in Computer Science 60, ed. R. Bayer, R.M. Graham and G. Seegmueller, Springer-Verlag, Berlin (1978).

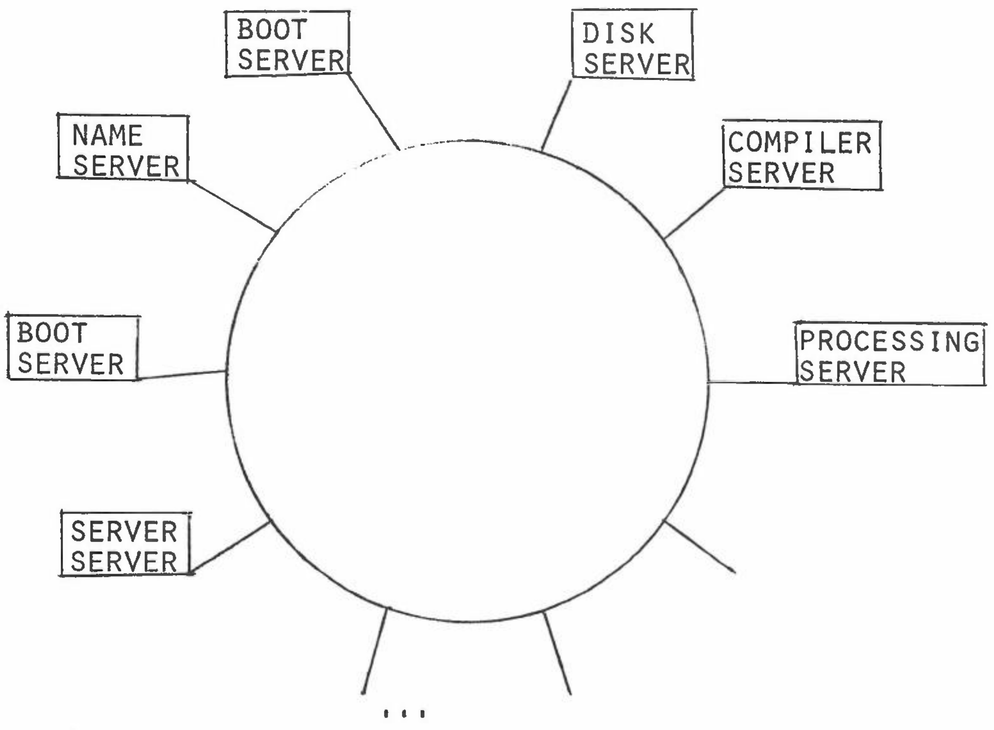
16. J. Goldberg, "SIFT: A Provable Fault-Tolerant Computer for Aircraft Flight Control," Proc. IFIP Congress 80, pp.151-161, North-Holland (1980).

17. J. M. Rushby and B. Randell, "A Distributed Secure System," Technical Report, Computing Laboratory, University of Newcastle upon Tyne (In preparation).

THE STRUCTURE OF

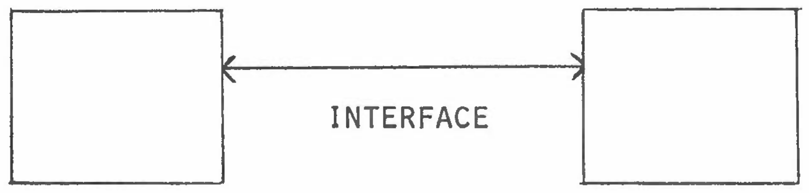
DISTRIBUTED COMPUTING SYSTEMS

A "STRUCTURED" DISTRIBUTED COMPUTING SYSTEM

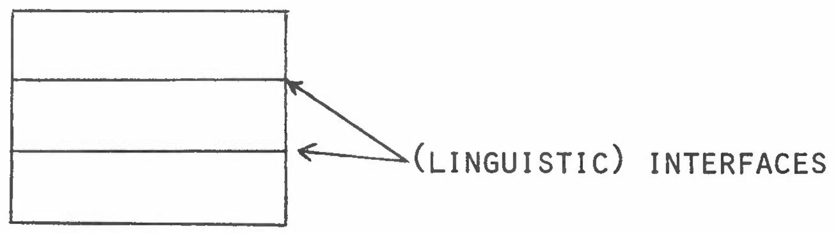


BASIC STRUCTURING TECHNIQUES

CO-EXISTING INTERACTING COMPONENTS



LEVELS OF ABSTRACTION



- GOOD STRUCTURING $\leftrightarrow$ "NARROW" INTERFACES
- SEPARABLE LOGICAL CONCERNS SHOULD BE IDENTIFIED, AND TREATED SEPARATELY

/DISTRIBUTEDNESS/FAULT TOLERANCE/HETEROGENEITY/SECURITY/ -

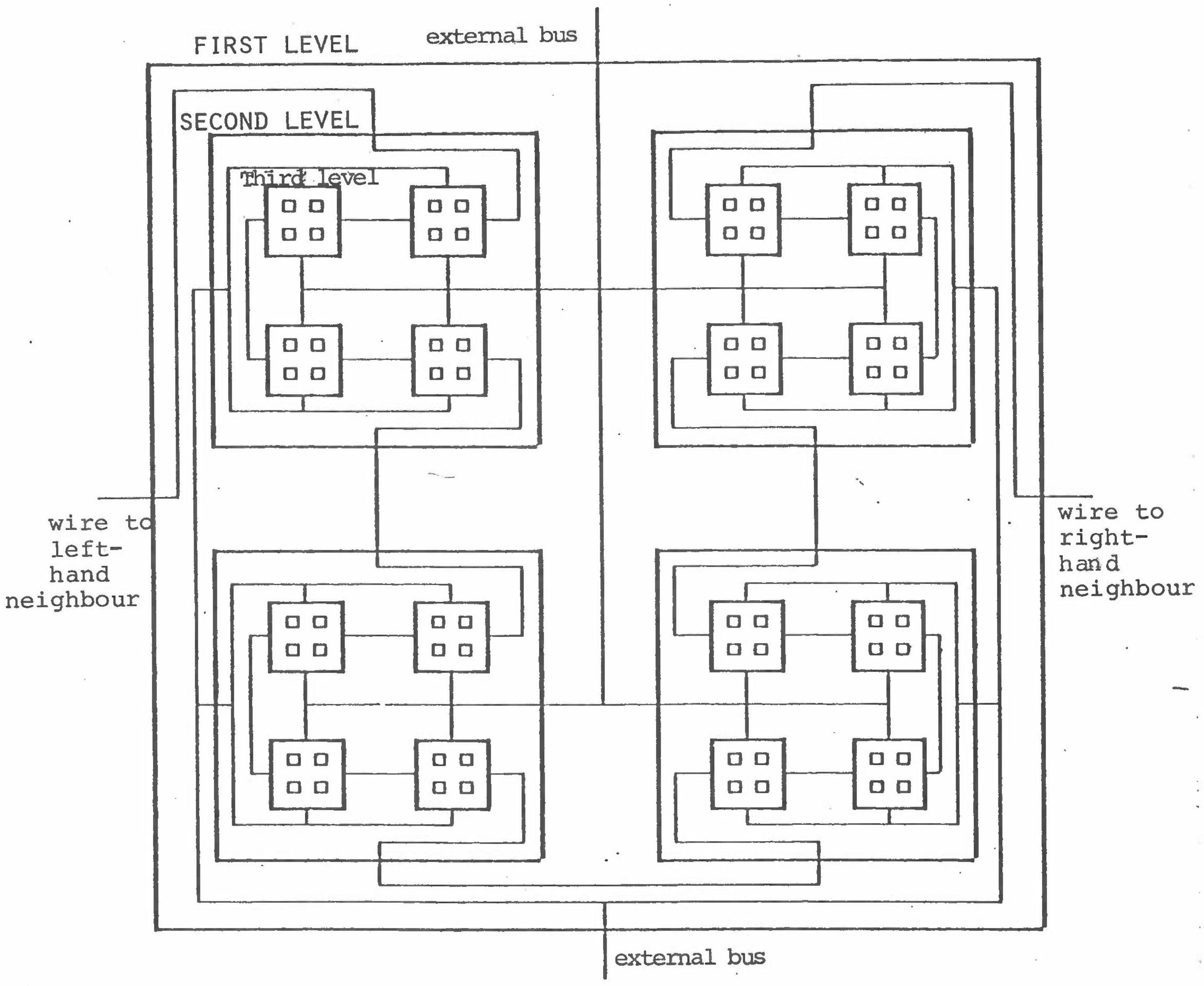
ARE ALL SEPARABLE!

A RECURSIVE STRUCTURING PRINCIPLE

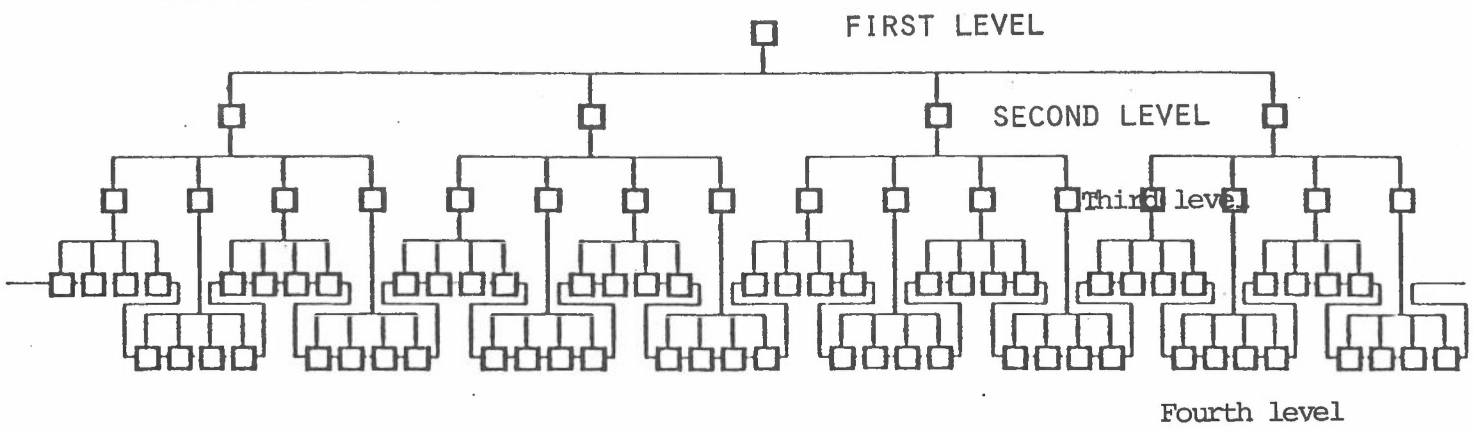
A DISTRIBUTED COMPUTING SYSTEM SHOULD BE FUNCTIONALLY EQUIVALENT TO THE INDIVIDUAL COMPUTING SYSTEMS OF WHICH IT IS COMPOSED, EVEN WITH RESPECT TO EXCEPTION REPORTING.

(IE THE INFORMATION THAT A SYSTEM PROVIDES TO ITS ENVIRONMENT WHEN IT IS UNABLE, OR PERHAPS EVEN NOT DESIGNED, TO CARRY OUT A REQUESTED OPERATION.)

RECURSIVE MACHINE, PHYSICAL SYSTEM ORGANIZATION:



LOGICAL SYSTEM ORGANIZATION:



ADHERENCE TO THE RECURSIVE STRUCTURING PRINCIPLE:

1. SIMPLIFIES THE TASK OF JOINING (EVEN DISTRIBUTED) SYSTEMS TOGETHER
   - VIZ BLOCK STRUCTURE IN PROGRAMMING LANGUAGES

2. MEANS THAT A SYSTEM IS INDEFINITELY EXTENSIBLE

BUT REQUIRES COMPONENT COMPUTERS TO PROVIDE:

1. (AT LEAST THE APPEARANCE OF) PARALLEL PROCESSING

2. CONTEXTUAL NAMING

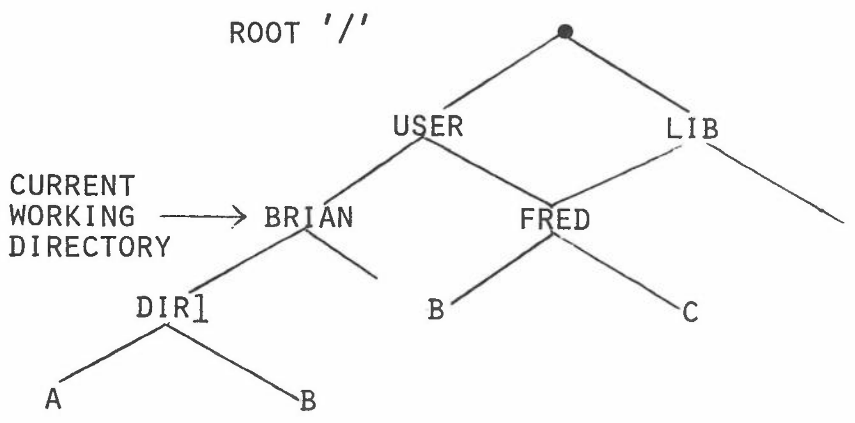
CONTEXTUAL NAMING

REQUIRES MEANS FOR INTRODUCING AND ENTERING (AND LEAVING) NEW NAMING CONTEXTS

AND THAT ALL NAMES FOR OBJECTS IN THE COMPUTING SYSTEM MUST BE CONTEXT-RELATIVE

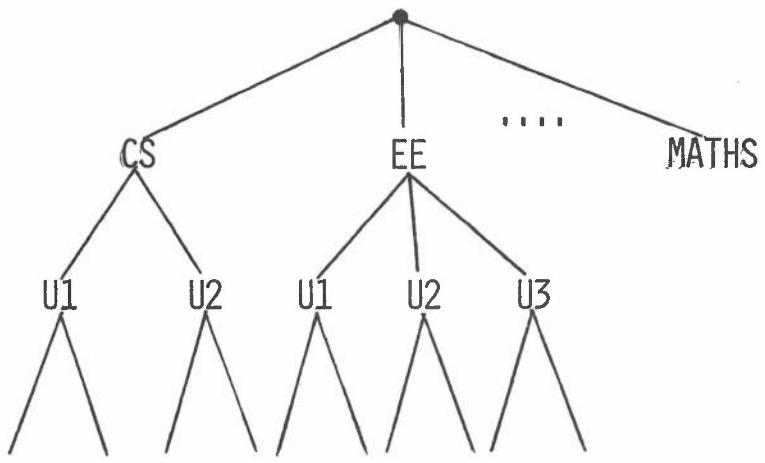
(A GIVEN SYSTEM NEED NOT KNOW WHETHER IT IS JUST A COMPONENT IN SOME LARGER SYSTEM)

THE UNIX CONTEXTUAL NAME SPACE



/USER/BRIAN/DIR1/A	≡	DIR1/A
/USER/FRED/B	≡	../FRED/B

A UNIVERSITY-WIDE UNIX UNITED SYSTEM



FROM WITHIN CS'S U1 MACHINE

/../U2 IDENTIFIES THE U2 MACHINE IN CS

/../../EE/U2 THE U2 MACHINE IN EE

(UNIX MACHINES ARE MADE TO LOOK LIKE DIRECTORIES, IE - CONTEXTS - IN A LARGER UNIX SYSTEM, "UNIX UNITED")

UNIX UNITED

- A DISTRIBUTED SYSTEM WHICH IS INDISTINGUISHABLE AT USER AND PROGRAM LEVEL FROM A UNI-PROCESSOR UNIX
- FILE ACCESS, DEVICE USAGE, I/O REDIRECTION, INTER-PROCESS COMMUNICATIONS, CHANGE DIRECTORY ETC ALL WORK ACROSS MULTIPLE MACHINES
- ALL ISSUES OF INTER-PROCESSOR COMMUNICATION AND OF NETWORK PROTOCOLS HIDDEN
- EXISTING PROGRAMS USING MULTIPLE PROCESSES CAN, WITHOUT BEING CHANGED, USE MULTIPLE PROCESSORS

DISTRIBUTEDNESS

(- THE PROBLEM THAT ARISES THROUGH HAVING MULTIPLE AUTONOMOUS COMPUTERS INTERACTING)

- A LOGICAL PROBLEM SEPARATE FROM, EG FAULT TOLERANCE

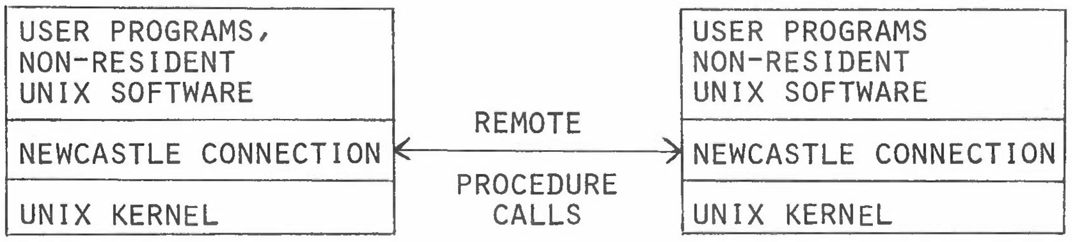
TWO PRINCIPLE ISSUES:

- ROUTING REQUESTS FOR ACTIVITY TO APPROPRIATE COMPONENT COMPUTING SYSTEM, FROM THE ONE IN WHICH IT ORIGINATED

- PRESERVING THE APPEARANCE OF A SINGLE (RECURSIVELY STRUCTURED) NAME SPACE, AND HIDING ANY LOCAL ADDRESSING

<u>THE NEWCASTLE CONNECTION</u>

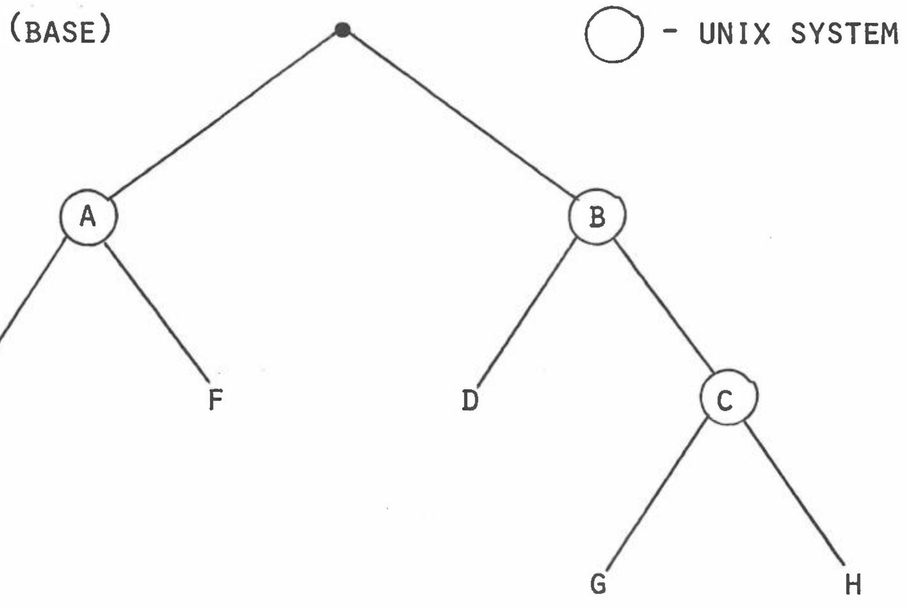
A TRANSPARENT SOFTWARE LAYER WHICH HANDLES ALL (AND ONLY) THE PROBLEMS OF DISTRIBUTEDNESS



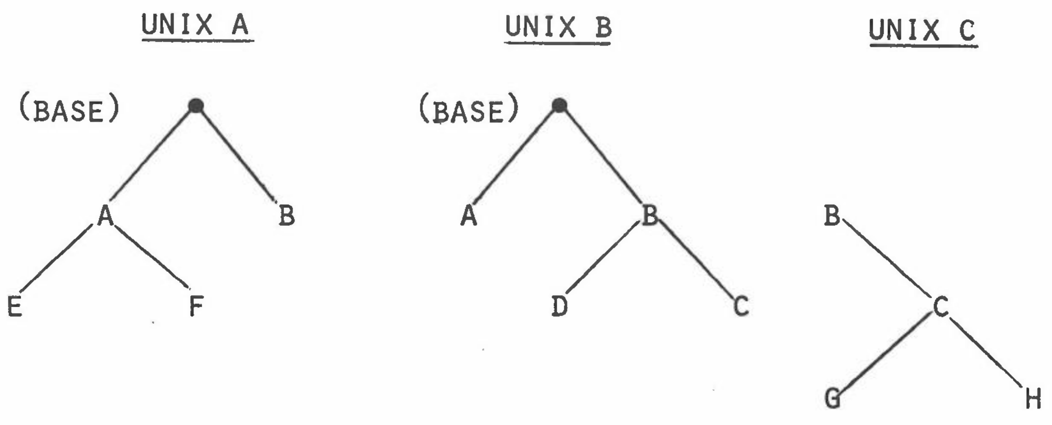
- FROM ABOVE, LOOKS LIKE THE KERNEL - FROM BELOW, LOOKS LIKE A SET OF USER PROCESSES

- NO CHANGE NEEDED TO EXISTING SOURCE CODE - UNIX OR USER PROGRAMS

UNIX UNITED

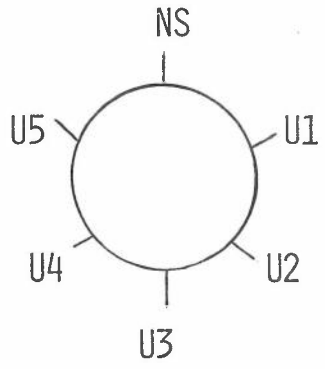


REPRESENTING THE NAME SPACE

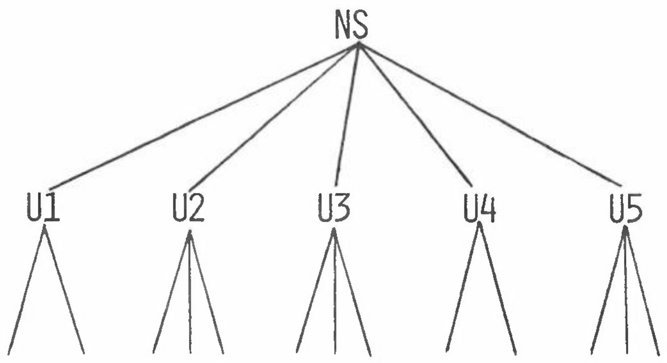


ACTUAL PORTION OF NAMING TREE (I-NODES) STORED ON EACH MACHINE

A NAME SERVER



COMMUNICATIONS STRUCTURE



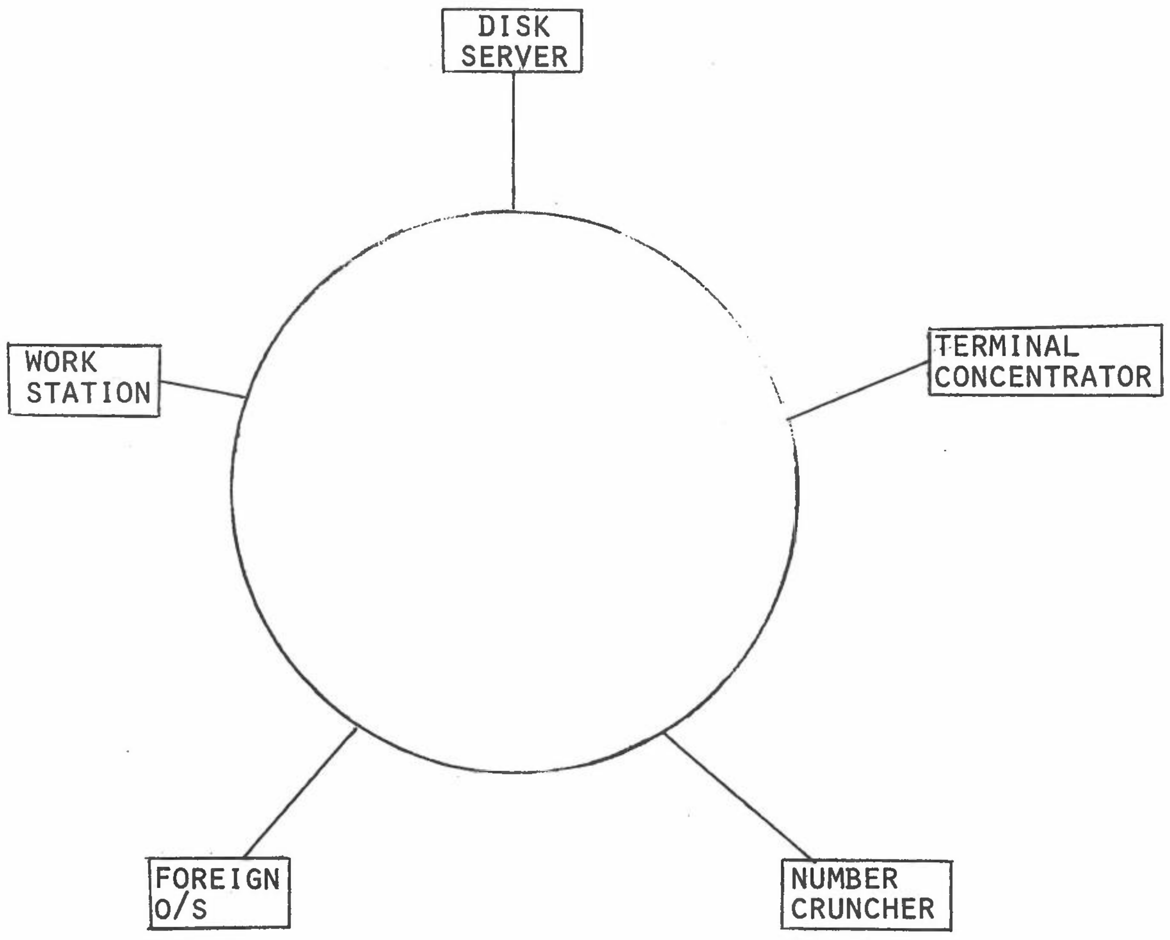
NAME STRUCTURE

NS CONTAINS HARDWARE ADDRESSES FOR ALL THE OTHER MACHINES. THEY EACH HAVE ONLY ITS ADDRESS (RING PORT NO.)

U1 ACCESSES FILES IN U2 USING /../U2 - PERMISSION TO OPEN A FILE INVOLVES NS, LATER READING AND WRITING DOES NOT.

HETEROGENEITY

THE COMPONENT COMPUTERS JUST HAVE TO APPEAR SIMILAR - AND CAN RETURN EXCEPTION MESSAGES TO SYSTEM CALLS THEY CANNOT SUPPORT



EACH BOX LOOKS LIKE A UNIX SYSTEM TO THE OTHERS!

"A VIRTUAL SYSTEM CALL PROTOCOL"

UNIX UNITED

VERSUS

OPEN SYSTEMS INTERCONNECTION MODEL

- UNIX UNITED HIDES THE BOTTOM THREE OF OSI'S SEVEN LAYERS OF PROTOCOLS
- AND LARGELY DOES AWAY WITH THE NEED FOR THE OTHERS - REPLACING THEM WITH UNIX'S SYSTEM CALLS

A FAULT TOLERANCE PRINCIPLE

FAULT TOLERANT SYSTEMS SHOULD BE CONSTRUCTED FROM GENERALISED FAULT TOLERANT COMPONENTS

(SUCH COMPONENTS TRY TO TOLERATE, WHEREVER APPROPRIATE, THEIR OWN FAULTS AND THOSE REPORTED TO THEM BY UNDERLYING COMPONENTS, AND ALSO BEING WRONGLY INVOKED BY THE COMPONENTS THEY INTERACT WITH AND THE COMPONENT OF WHICH THEY THEMSELVES FORM PART.)

FAULT TOLERANCE

1. DISTRIBUTED SYSTEMS PROVIDE NEW OPPORTUNITIES FOR OBTAINING HIGH RELIABILITY THROUGH FAULT TOLERANCE

   PLUS

2. NEW TYPES OF FAULT

3. THESE ARE TWO SEPARATE ISSUES, AND ARE ALSO SEPARATE FROM ANY OPPORTUNITIES OR REQUIREMENTS FOR FAULT TOLERANCE THAT WOULD EXIST IN AN EQUIVALENT CENTRALIZED SYSTEM.

THE NEWCASTLE CONNECTION DEALS WITH ITEM 2 ONLY.

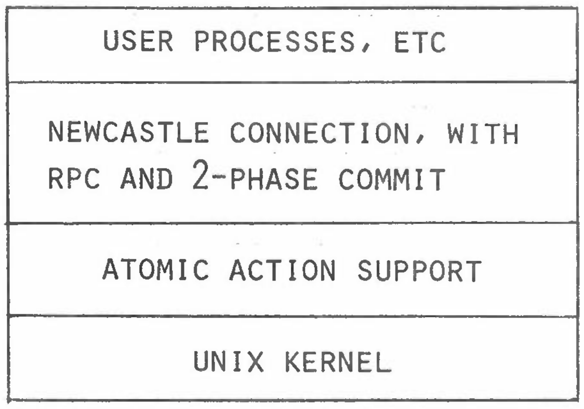
FAULT TOLERANCE WITHIN THE CONNECTION

THE CONNECTION USES A REMOTE PROCEDURE CALL PROTOCOL TO ACHIEVE "EXACTLY ONCE" SEMANTICS;

- TO GUARANTEE THAT SYSTEM CALLS ARE NOT ACCIDENTALLY REPEATED BECAUSE OF THE CALLED MACHINE CRASHING AND RESTARTING, OR COMMUNICATIONS PROBLEMS.

- IT DOES NOT CONCERN ITSELF WITH PROBLEMS ARISING FROM CRASHES OF THE CALLING MACHINE - THIS IS WHAT ATOMIC ACTIONS ARE FOR - AND THEY ARE EQUALLY NECESSARY ON A TIME-SHARING MACHINE (IE ORDINARY UNIX)

FAULT TOLERANCE LAYERS



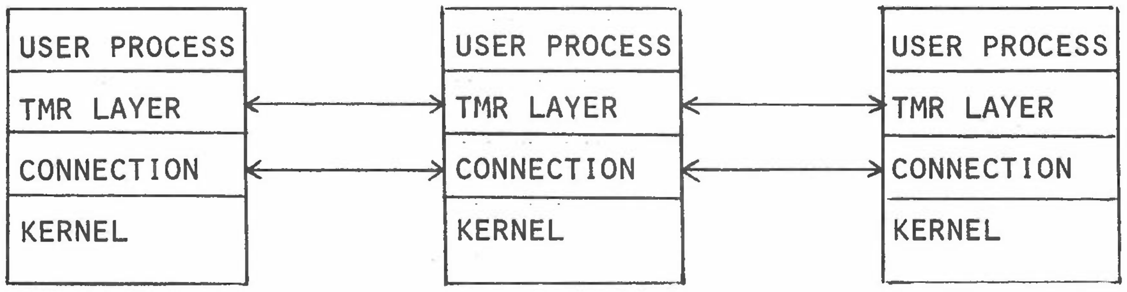
ATOMIC ACTION SUPPORT PROVIDES EXTRA SYSTEM CALLS

ESTABLISH RECOVERY POINT
(PREPARE TO DISCARD RECOVERY POINT)
DISCARD RECOVERY POINT
RETURN TO RECOVERY POINT

SLIGHTLY EXTENDED NEWCASTLE CONNECTION MAKES THEM WORK FOR A DISTRIBUTED PROCESS.

MASKING HARDWARE FAULTS

THE CONNECTION LAYER HIDES THE LOCATION OF PROCESSING AND STORAGE - A FURTHER (SIMPLE) TASK TO HIDE THEIR REPLICATION



- TMR LAYER PERFORMS SYNCHRONIZATION, MAJORITY VOTING, AND RECONFIGURATION

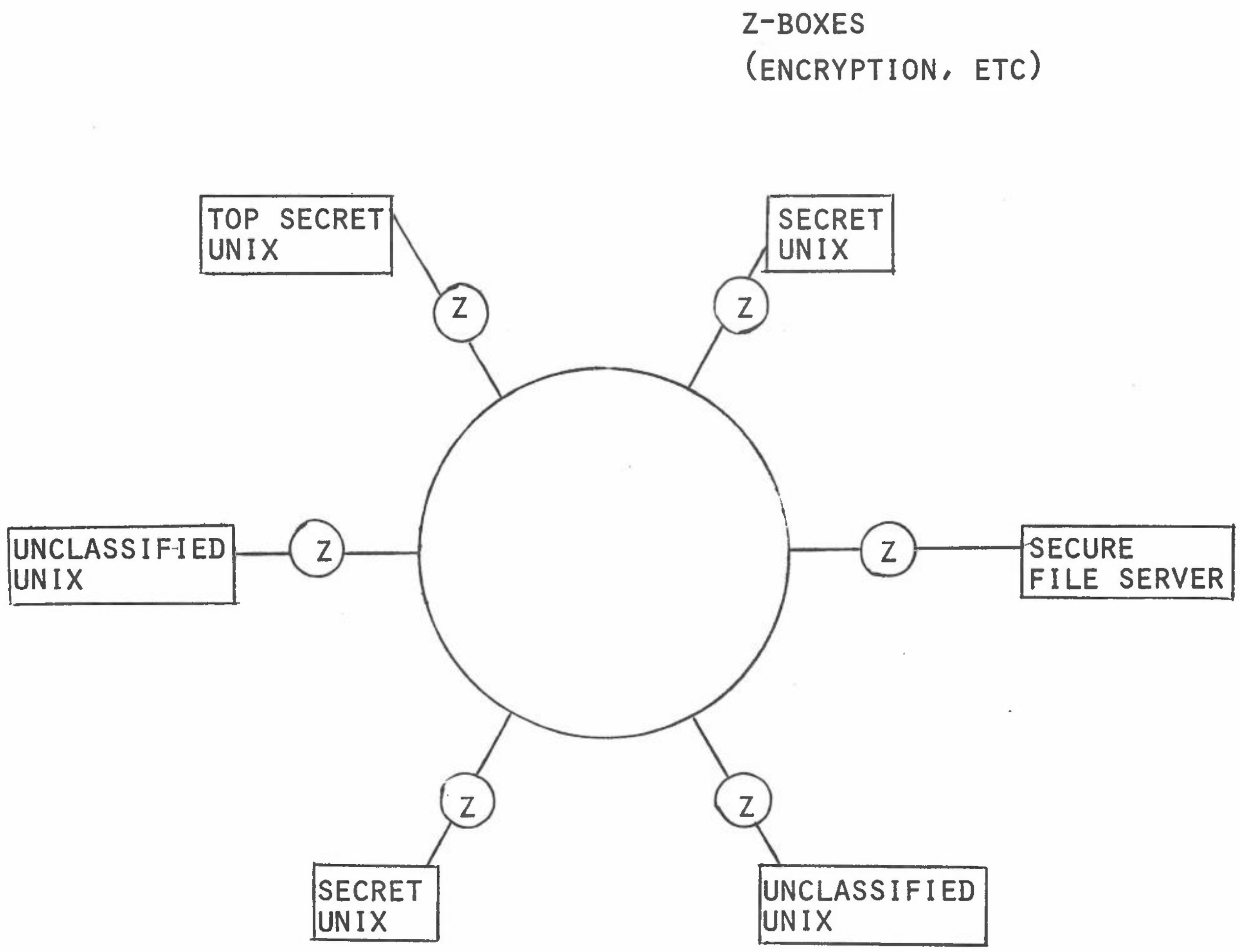
- USER PROGRAM UNCHANGED - BUT PROVIDED WITH HIGHLY RELIABLE HARDWARE

SECURITY

- UNIX UNITED ALLOWS EACH UNIX SYSTEM TO HAVE ITS OWN ADMINISTRATOR, STILL IN CONTROL OF ACCESS TO HIS MACHINE
- NEVERTHELESS, A USER ONLY LOGS IN ONCE, TO WHOLE SYSTEM, AND IS THEN SUBJECT TO NORMAL UNIX ACCESS CONTROLS

- MILITARY-TYPE SECURITY REQUIRES FORMAL CERTIFICATION OF SECURITY MECHANISMS - EVEN UNIX MUCH TOO COMPLICATED
- THE RECURSIVE STRUCTURING PRINCIPLE ALLOWS ONE TO SUBDIVIDE, AS WELL AS TO COMBINE, SYSTEMS

MULTI LEVEL SECURE UNIX



CONTRUCTED FROM

TRUSTED COMPONENTS - FOR PREVENTING INFORMATION FLOW, AND FOR MONITORING/MEDIATING PERMITTED INFORMATION FLOW. (SIMPLE ENOUGH TO CERTIFY)

UNTRUSTED COMPONENTS - GENERAL PURPOSE UNIX SYSTEM, WORKING AT A SINGLE SECURITY LEVEL

IN TOTAL, LOOKING LIKE A SINGLE CONVENTIONAL UNIX SYSTEM!

THE STRUCTURING PRINCIPLES:

GREATLY SIMPLIFY CONSTRUCTION OF DISTRIBUTED SYSTEMS, BY ALLOWING MANY ISSUES TO BE DEALT WITH SEPARATELY

THIS IS NOT JUST THEORY:

- PHASE 1 UNIX UNITED (REMOTE FILE AND DEVICE ACCESS) BUILT IN FOUR WEEKS
- REMOTE EXECUTION, MULTIPLE USER GROUPS, ETC ADDED IN FEW MONTHS
- PROTOTYPE MULTI-LEVEL SECURE UNIX CONSTRUCTED IN LESS THAN A WEEK
- TMR-UNIX IN A FEW MONTHS
- SYSTEM NOW BEING MADE AVAILABLE TO OTHERS

CLOSELY COUPLED SYSTEMS

R. L. Grimsdale
University of Sussex.

The first part of this paper describes the general concepts of closely-coupled systems and the second part reports on some examples of closely-coupled systems which have been developed in the U.K.

PART I CONCEPTS

Introduction

A closely-coupled system may be defined as a collection of linked processor and memory modules which collaboratively execute a single job under one overall management. A loosely-coupled system, on the other hand, is a set of linked computers each under the control of a separate user and each executing a distinct program, but with the several computers communicating or making access to a common database from time to time. An example of a loosely-coupled system is a collection of personal computers distributed around a building linked together by a communication channel. The individual computers may be used for quite different purposes. The programs or users may send messages to one another or access a shared filestore. There is no central management although certain standards for data exchange must be adopted.

In our definition of a closely-coupled system there is an underlying concept of a single activity, sub-divided into a collection of tasks which execute in parallel. At certain periods in their activities, the tasks inter-communicate in a well-disciplined way. A task is organised to send a message to a second task, this second task being organised to receive such a message from the first task.

Taking an overview of the closely-coupled system, it is seen as a collection of processor, memory and input-output modules with a certain inter-connection scheme. Such a hardware arrangement requires the addition of system software to make it useable. Traditionally, this support software takes the form of an Operating System and a Programming Language. A basic operating system comprises modules or device drivers which hide the fine details of the operation of devices such as printers and disc memories. A more advanced operating system will permit several user programs to share the hardware facilities, giving each user the impression that he has exclusive use of a private or virtual machine. A further facility which may be provided is a mechanism to manage movement of blocks of data between discs and the immediate access memory. This can include a filing system whereby data blocks can be encapsulated in named files, with a directory being provided for the user.

The programming language simplifies the programming task and modern programming languages provide a structured framework for good program construction together with a versatile naming facility for the objects to be manipulated. The system is layered. The lowest level is the hardware, upon which the operating system creates virtual machines and then, at a further level, the programming language compiler effectively converts the system into a machine which acts as an interpreter for the high-level language application programs.

Closely-coupled systems normally are embedded, that is they form part of a larger engineering system, for example, industrial process control, air traffic control or telephone exchange control system. The system designer must be provided with good facilities to program the system for his particular application. This must include adequate access to certain hardware features not normally available in a high-level language. It will be seen that a high-level language for a multi-processor system can include features normally provided in the operating system. Comprehensive control of the system is therefore available through the high-level language.

There are various motivations for adopting a closely-coupled system architecture. An increase in speed over that available in a single processor can be achieved if the system algorithm can be decomposed into tasks which can operate in parallel. Redundancy can be exploited to provide improved reliability or availability. Perhaps the main reason for adopting a multiprocessor approach is because the system itself is inherently distributed. Processing power can be directly associated with each of the main activities, with data movement being effected between the functional modules.

Architecture

The technique used to interconnect the processor modules influences the characteristics and performance of the system. The simplest arrangement is that in which one processor has an output port and another has an input port; these are connected, so that a word sent to the output port will appear on the associated input port. The ports can be replaced by a register shared by the two processors. A hardware semaphore mechanism is necessary to ensure that the writing and reading operations on this register are mutually exclusive. This arrangement can be extended to become a multiplicity of registers by use of a shared memory module. Each processor connected to the shared memory via a port and a hardware mechanism is provided to prevent simultaneous accesses. The meaningful use of the shared memory is a software consideration.

A number of different topologies are used. A module can have a connection to one, two or four neighbours or to many. The two-neighbour connection implies a ring configuration. An alternative arrangement is that in which a shared bus is employed. Finally, there is the one-to-many, or broadcast configuration. In the direct connection between modules, data is transferred under the control of a mechanism which ensures exclusive access to the data. In the ring system, data can be sent from any module to any other on the ring in the form of a short packet. The ring operates under the control of a token which is a short, particular bit sequence. A module that wishes to transmit a packet may do so when it observes the passage of the token. The transmitting module modifies the token, converting it to a connector and then injects the packet into the ring. At the receiving module the packet is marked by an acknowledgement and completes its journey around the ring back to the sender. The token ensures that there is no contention for the ring and the return of the packet to the sender provides the necessary acknowledgement mechanism. The ring resorts to a contention system if the token is lost. Any module may then attempt to inject the token and arrangements must be provided to ensure that only one token remains on the ring. The shared bus relies on contention resolution for its operation. In the Carrier Sense Multiple Access Collision Detect (CSMA/CD) system a module will wait for silence on the bus and will then attempt a transmission which, if successful, will be acknowledged by the receiver. However, it is possible that a second module may transmit simultaneously with another module, possibly because the second module initiated its transmission before it received the transmission from the first module. In this case, the modules may hear each others transmissions, but if not they become aware of the collision because the messages are corrupted and the receiving modules do not send acknowledgements of an error-free reception. The transmitting modules will make further attempts after random time intervals.

The particular form of closely-coupled systems that we are concerned with here are those in which parallel execution occurs at the task level and in which each task is a significant segment of sequential program code. This is in distinction from the dataflow class of machine in which parallelism occurs at the instruction level or distributed array processors in which identical operations are performed on many operands in parallel.

Concurrency Support Mechanisms

Certain basic requirements to support concurrency can be identified. Firstly, it is necessary for the results produced by one task to be used as an input to another task, and therefore some satisfactory mechanism of data transfer between tasks must be provided. Secondly, there is normally a need to synchronise the operation of tasks. Because of mutual inter-dependence, one task cannot proceed beyond a certain stage without some action occurring in a dependant task. Similarly the dependent task cannot continue beyond a particular stage until the previous task has accomplished certain operations. Summarising, there is a need to pass data between tasks, and a requirement to synchronise tasks.

Before the advent of programming languages which supported concurrency it was necessary to employ a multi-tasking operating system to perform the inter-tasking operations. This provided a way of binding together a number of separate sequential programs to form a multi-task system. A common arrangement was to provide a bounded buffer for inter-task communication. The operation *put(item)* and *get(item)* respectively stored a data item in the buffer and retrieved an item from the buffer. The functions *full* and *empty* were used to test the current state of the buffer. To prevent simultaneous access to the buffer it was necessary to employ a semaphore (1). There are objections to the use of this technique because the primitives can be easily misused or even omitted. The problem arises because the primitives are at too lower level. Because of this, various high-level constructs have been incorporated in programming languages, having the advantage of a degree of compile-time protection.

The Monitor Concept

The Monitor (2,3) is an arrangement in which shared data is encapsulated with a set of procedures which perform operations on that data. The data cannot be accessed except through the use of these procedures. The monitor is activated by a process (external to the monitor) making a call on one of the monitor procedures. Only one process at a time may be actively executing a monitor procedure within a given monitor. The monitor construct therefore provides exclusive access to shared data and has the advantage of encouraging careful programming of the monitor procedures thereby safeguarding the monitor data structures against misuse. If no process has made a call on a monitor procedure, then when a process makes a call on a procedure of that monitor, that call will be serviced immediately. The execution of the calling process will be suspended until the procedure has completed its operation and returns control to the calling process. However, if a monitor procedure is in execution, then any other call on a procedure of that monitor will not be serviced immediately, the calling process will be suspended and the request will be placed in a queue. When the activated procedure completes its execution, the process which called it leaves the monitor; the queue is inspected, the waiting request is serviced and on completion of the associated procedure call, the calling process leaves the monitor and resumes its independent execution. In this way, the processes access the shared data in mutual exclusion.

The monitor thus provides a mechanism for accessing shared data in an exclusive and therefore safe manner; it does not directly synchronise the processes which call it, but provides a signalling mechanism whereby this may be accomplished. A process which gains access to a monitor procedure can, within the procedure issue a *wait* signal. The process is then suspended and another process is allowed to enter the monitor. The waiting process will be resumed when another process enters the monitor and sends a signal for that waiting process. The monitor is an elegant concept and can ensure that the states of the processes which use it are deterministic. It can be implemented very effectively on a multi-processor system which uses shared memory since it is inherently a mechanism for gaining controlled access to a block of shared data. It is not so convenient to use in a distributed system.

The Rendezvous

The requirements for inter-process communication are, firstly, a mechanism for data exchange, which in the monitor system is provided by the sequential use of a shared memory by the communicating processes. The second requirement is for a scheme for synchronising processes. This need for a synchronising mechanism arises because processes must keep in step with one another. This is exemplified by the case in which a producer process generates a sequence of values which are to be subsequently processed by a consumer process. The consumer process must accept values from the producer, one by one, such that the rate at which the values are consumed matches the rate at which they are generated by the producer process. If the two processes were free running they could get out of step. The relative timing of the processes must therefore be enforced by synchronisation, which implies that the faster process must be caused to wait for the slower. The rendezvous mechanism, introduced by Hoare (4) and Brinch Hansen (5) combines the operations so data transmission and synchronisation is performed in one mechanism.

In the scheme as introduced by Hoare (4), if a process A wishes to transmit data to a process B then each process must announce its intention to communicate. Process A will include within the sequence of instructions it executes, a request to transmit to process B. Similarly process B includes in its sequence a request to receive from A. If x is the datum to be transmitted from A and y is the name of the variable (or location) in B which is to receive it, then process A includes the statement

B ! x

and process B has the statement

A ? y.

If process A executes the statement B!x first it is suspended until B reaches the statement A?y. Similarly, if B arrives at the request first it will be caused to wait until A arrives at its request. When both processes have reached the rendezvous the data is transferred and the processes resume their respective executions. The mechanism is symmetric, in that the caller announces the name of the receiver and vice versa. This symmetry is not practical if the receiver is a library process which might be required to be called by several processes unknown to it.

The alternative, proposed by Brinch Hansen (5) and adopted in Ada is asymmetric; in this the caller announces the name of the server (receiver) process, but the callers remain anonymous to the receiver. The server process includes an *accept* statement within the body of its code. This accept statement is very similar to a procedure statement. The reserved word *accept* is followed by the name of the entry and after this comes the list of formal parameters. Finally, there is the sequence of statements which is executed when the rendezvous occurs. Through the use of parameters, data can be passed from the caller to the called process and vice versa. It is not always convenient to require that a called process should only respond to a particular entry call. A non-deterministic arrangement has therefore been introduced whereby a called process can at a particular point in its operation respond to a number of different entries. The classical example is the case in which the called task is the manager of a bounded buffer; the alternative responses it must be able to make are to receive an item into the buffer or to deliver one. Furthermore, guards can be associated with the alternatives so that, for example, a request to deliver an item will only be serviced if an item is present in the buffer.

For specific applications, less general constructs may be useful. For example, a system based on message passing may be appropriate. In this a number of bounded buffers can be created which are used as carriers of messages between processes. For some process control or real-time data processing applications a broadcast facility is desirable. This will be non-deterministic because there will be no attempt to ensure that all receivers have received the message, nor will there be any attempt to check that a particular receiver has received all the messages that have been sent. Although this transmission system appears unsatisfactory, it may be acceptable for the particular application because the same information may be sent repeatedly from a sensor and it is unimportant if an individual value is lost.

PART II EXAMPLES OF CLOSELY-COUPLED SYSTEMS

The CYBA-M Multi-Microprocessor - U.M.I.S.T.

The CYBA-M multi-microprocessor (6) originated at the University College of Swansea and is now at the University of Manchester Institute of Science and Technology. The objective was to provide a design method and development environment for the implementation of multi-microprocessor based products. It was assumed that, in the target systems, the overall algorithm could be suitably sub-divided into a set of concurrent tasks. The programs to execute the individual tasks would be held in the ROM of the individual processors. There would be static interconnections between the individual processors. CYBA-M was developed as a vehicle which would enable experimentation on those general aspects of concurrent processing element design which are largely device independent.

CYBA-M is based on a shared memory architecture and consists of 16 processor elements, each of which comprises an 8080 processor with a 16 Kbyte local memory. Each processor element has a connection to a port of a 16-port 10 Mbyte/sec global memory. The global memory is used to provide inter-connections between the processing elements. The relatively high data rate of the global memory was selected to minimise the contention on that memory. In addition, the processing elements can access a 16-port image memory which provides memory-mapped connections to peripheral devices.

The software approach (7) which has been adopted for CYBA-M is based on concepts of control-flow and data-flow. These techniques have been used to formulate parallelism, including pipe-lining and over-lapping. Recent work has concentrated on the exploitation of CYBA-M for a number of real-time applications.

CONIC - Imperical College of Science and Technology, London

CONIC (8) provides an integrated set of techniques and tools for constructing and managing large distributed computer control systems, particularly for industrial process control. The software architecture is modular and systems can be configured from multiple instances of the modules. The module is defined as the smallest replaceable software component of the system and consists of a set of concurrent tasks. The tasks forming a module always reside in a single physical station, but it is possible to have more than one module in a station.

Both modules and tasks within modules inter-communicate by passing typed messages. Entry and exit ports are defined for the modules. A system is configured by creating instances of modules at stations and by linking the entry and exit ports of those modules. This configuration can be done at run-time, to allow for extension and modification of the system. The Conic language provides two types of communication primitives. The first type is a set of *request-reply* primitives which are intended to be used to send commands and receive responses or to query the status of other components in the system. These are synchronous in operation in that the sender is blocked until a reply is received from the responder. Also the responder can be blocked whilst awaiting requests to arrive. The second set of communication primitives provide unidirectional message passing which meets the requirement for the transfer of alarms and status information. The sending task is not blocked, since no reply is expected. Because it is not blocked it could update the message whilst it is in the process of being delivered. Accordingly, a buffer of a pre-determined fixed size is employed. When the buffer is full the oldest message is over-written.

Message passing over the network is performed by a communication system which supports two main classes of service. These are, firstly, the virtual circuit which provides an error-correcting link between stations. The second is a datagram service, without error-correction.

To date, the kernel which implements multi-tasking and inter-task communication, the communications system and operating system which provides facilities for loading and modifying the system have been implemented on a system composed of 5 LSI-11 computers connected by asynchronous links to form a communications ring.

DEMOS - National Physical Laboratory and Scicon

The DEMOS system (9) has a theoretical capability for up to 254 mini- or micro-computers, but typically the number would be in the range 5 to 50, each connected to a 16-bit parallel ring which is clocked at 10MHz. The individual computers cannot access each others memories; communication between them is implemented by message transfer over the ring. DEMOS is designed to be programmed as a single virtual machine in Concurrent Pascal (10). This language uses Monitors to support concurrency. An applications program consists of a number of modules which, when compiled, are statically allocated to processors by a systems generation program. Inter-computer communication is managed by a kernel resident in each processor. Information is transferred in blocks over the ring which functions as a multi-ported block transfer mechanism. It was recognised that shared memory architectures are better suited for Concurrent Pascal and that network architectures of the type selected suit a non-deterministic message passing language. Nevertheless, it was considered that the use of *monitors* and *classes* in Concurrent Pascal provided a better programming environment and that the overall design was a reasonable compromise.

It was necessary to devise a scheme for operating the monitor concept in conjunction with the ring connection between processors. Because a process attempting to access a monitor may be delayed, (by virtue of the mutually exclusive access to a monitor), messages containing the parameters of a call on a monitor procedure may have to be queued. Queuing is not done at the destination kernel because this would require too much space, and therefore the technique adopted is to delay the transmission of a message until the successful completion of a handshake protocol conducted by the kernels.

POLYPROC - The University of Sussex Multi-Processor System

POLYPROC (11) is an experimental multi-microprocessor system which serves as a vehicle for exploring techniques for constructing embedded systems. Particular emphasis has been placed on providing a good environment for systems design through the use of a high-level language which encompasses the features of an operating system.

The programming language which has been implemented on Polyproc is MARTLET (12). This language is based on sequential PASCAL with the incorporation of certain features for inter-process communication, notably the rendezvous concept similar to that used in ADA (13). The aim was not to implement the full Ada language but rather to incorporate the more important features which support concurrency. Thus Martlet includes many of the structural features (task modules) and the concurrency features (entry-call/accept, select). In addition, instancing and exception handling are incorporated.

The methodology adopted for the development of Martlet programs for Polyproc allows the application programmer to create a suite of task modules without the need to know about the particular hardware configuration on which those tasks are to run. The task modules can execute concurrently and communicate with each other in order to perform the overall system function (14). The task modules can be compiled separately and the compiled code for each module can be statically assigned to the various processing elements which comprise the selected hardware configuration. This is known as the system configuration phase, and is, in turn, performed by a system configuration program. This latter is an interactive program which, in response to a dialogue with the operator, takes the compiler output and in turn produces a memory image for each local processor within the system together with the necessary data structures to be used by the run-time support software to effect inter-task communication.

The compiler and the configuration program are both written in Pascal to allow portability and run on a host computer which is linked to the Polyproc system. The host machine also includes performance monitoring and diagnostic software which can be employed to facilitate the location and run-time errors during the development phase and to obtain information about the run-time performance of the system.

The Polyproc Härdware Configuration

The hardware configuration for a Polyproc system is very flexible and is not restrained to the use of a particular microprocessor. The particular architecture to be described is that which has been implemented in the laboratory, but as will be seen later, the cost of adopting different forms of architecture is low.

Ultimately as processors become cheaper it will be possible to implement each particular computing function or task by an individual processor or station. At the present time, costs are such that it is still economical to include an element of time sharing in the allocation of tasks to processors.

A Polyproc system consists of one or more Stations each of which consists of a number of processing elements with associated local (private) memory. Also, processing elements in a station have access to a shared or global memory which is used as a means for fast communication between processing elements within the same station.

In recognition of the fact that the control of the inter-tasking operations within a station could usefully occur in parallel with the execution of the tasks themselves, it was decided to include a Control Processor within each station. This control processor does not execute any application software but serves to schedule the execution of tasks on the several remaining processors in the station which are referred to as Local Processors. Inter-station communication is under the management of a separate control processor. In the current system CSMA/CD serial data transmission is used for inter-station communication, employing a single coaxial cable to which each station is connected by a tap. A station is composed, therefore, of a control processor, a communications processor and a local processor, the number of the latter being chosen to suit the applications requirement.

The processor adopted for the system is the Intel 8086, and a station consists of a number of SBC 86/12A single board computers which plug directly into the main station bus. This latter is based on the Intel Multibus, and additional shared memory and peripheral interface boards also plug directly into the same bus. Each processor board contains an 8086 CPU, an amount of local memory (PROM and RAM) and various peripheral interfaces and timers.

Run-time Support Software

The Martlet compiler generates an intermediate language, M-Code, which is executed interpretively. The use of an intermediate language was adopted because it was not justifiable to devote effort to the production of a translator to generate machine code for the target machine. Although interpretation is slower than full translation, the approach has a number of advantages. The software can be transported to different target machines at very little cost, the intermediate code is very compact and it is possible to incorporate instrumentation mechanisms to provide information about the performance of the system.

As described above, each Polyproc station consists of a control processor, a communications processor and a number of local processors. The control kernel (15) is contained within the control processor and each local processor has a small local kernel. The logical function of the control kernel is to transform the system hardware into a virtual machine which supports the concurrency features of the language. As will be explained later, the control kernel of one station interacts with the others to support inter-tasking operations between tasks in different stations as well as supporting communication between tasks in the same station. To facilitate readability and permit modifications in the development of the system, the control kernel has been written in Martlet. The control kernel is the sole owner of all vital and critical data structures for the management of the station operation.

The control kernel data structures consist of a suite of task activation records (one per task module resident in the station), a set of processor status records (one per local processor), a station directory containing the names and locations of each task, an interrupt map table to allow interrupts to be mapped into calls to the correct service task, and a set of interrupt and communication channel flags, the latter being used to control inter-station communication. A task activation record is created and initialised at the time a task is assigned to a specific local processor. In addition to information such as the task's name, its priority, and processor identity, the task activation record also includes data that are used by the control kernel to implement communication primitives at run-time. The processor status record holds information about the task currently being executed, the task next to be executed, and an array of queues holding tasks that are ready to run on the processor.

To illustrate the mechanism which is invoked to achieve a rendezvous, the detailed sequence of events will be described for the case in which both the calling task A and called task B are in the same station:

(i) The calling task A first evaluates the value and variable parameters for the call. This action is analogous to a procedure call in a procedural language. These parameters are evaluated in the local memory space of the local processor on which A is executing. These parameters are then copied into the parameter area designated for it in the station shared memory. Task A then issues the entry call.

(ii) The local kernel of the local processor on which A is running then places the address of the called task, together with the index of the entry to be called and the pointer to the parameter area of the calling task, in the mailbox of the caller's task activation record. The local kernel then requests the control kernel to switch the calling task.

(iii) The control kernel which has been running concurrently on the control processor has been polling the requests and on detecting the request in (ii) the control kernel suspends the calling task and links it to the entry in the called task.

(iv) When the called task reaches the rendezvous point (or it may already have reached this point and hence be suspended), the local kernel of the local processor on which the called task runs, pushes the address of the parameter block of the caller onto the called task's stack. The called task then uses this pointer to access the parameters, executes the entry sequence and returns values via the variable parameters before issuing the end-accept call. The local kernel relays this end-accept request via the mailbox of the called task.

(v) The control task implements an end-accept request by first releasing and rescheduling the calling task and then rescheduling the called task.

(vi) When task A runs again it then copies the variable parameters (if any) back into its local memory space.

(vii) Finally, task A and task B continue running concurrently.

The copying overheads while passing parameters is relatively low due to the fact that only the pointer to the parameter area in the shared memory need be passed to the called task. The rendezvous mechanism can therefore be implemented relatively efficiently on a shared-memory multiprocessor architecture.

In the case when the calling and called tasks reside in different stations two approaches are possible. In the dynamic method, inter-station communication must be set up at the time of the entry call, whereas in the static method provisions are made in advance.

In the dynamic approach (16), no memory is pre-allocated and a three-part protocol must be used to handle each inter-station entry call. The control kernel in the calling station, on receipt of any entry call to a task in a different station, sends a *request-for-entry* message to the destination station. The control kernel in the latter then inserts the request in the appropriate entry queue to wait for the called task to become ready for the rendezvous on this entry. It may, however, be already waiting. When ready, a *ready-to-accept* response is returned to the control kernel in the calling station, which, in turn, responds by sending the parameters associated with the call. Similarly at the completion of the rendezvous, any variable parameters associated with the call are returned to the calling task. This dynamic approach will obviously necessitate dynamic allocation of memory for the parameter area of the tasks involved in an inter-station communication. Furthermore, system performance is degraded because of the extra burden placed on the control kernel as it assumes responsibility for all communications with tasks in other stations.

The alternative static approach, which is the one which has been adopted in the Polyproc system, overcomes the problems of dynamic memory allocation at the expense of making a pre-allocation. In this method, a pair of intermediate transport tasks are introduced automatically at system configuration time. The two intermediate tasks then act as local agents - one for the caller and the other for the called task. The operation is then transparent to the control kernel as it is the transport tasks which handle the interface to the low-level communications protocol. This functional separation permits a different low-level communications layer to be substituted if desired. There are no run-time memory management problems as the required buffer memory is allocated in advance at system generation time, but the static approach can be expensive in memory if there are a large number of possible inter-station calls. There are, inevitably, greater overheads involved in making inter-station calls, compared with those between tasks in the same station, when shared memory can be employed.

Interrupt Handling

It is desirable to handle interrupts in a way which is both structured and independent of the specific hardware employed. The interrupt is regarded as an entry to a task which includes an accept statement for that interrupt.

To implement this scheme in a way which is compatible with the requirements of the hardware, it is necessary to have a low level interrupt handler on the local processor to which the interrupt source is physically connected. On receipt of an enabled interrupt, the task currently running on that processor is interrupted. The low-level interrupt service routine sets a flag in the shared memory which is subsequently polled by the control kernel of the station. The control kernel, on detecting the set flag, creates a dummy task activation record and links it to designated entry of the interrupt service task. The interrupt service task is then scheduled, and when it runs the interrupt request is serviced. Obviously arrangements can be made to poll interrupts which require fast response times, more frequently.

Experience with Polyproc (17) has shown that the rendezvous inter-task mechanisms are convenient both to use and to implement. They provide a one-to-one and a many-to-one form of communication, but do not support a broadcast requirement directly. As expected, inter-task communication within a station, supported by shared memory, is efficient, but overheads are encountered when the communication is between tasks resident in different stations.

REFERENCES

1. Dijkstra, E. H., Co-operating Sequential Processes in Programming Languages, Ed. F. Genuys, p. 43-111, Academic Press, 1968.

2. Hoare, C. A. R., Monitors: An Operating System Structuring Concept, Comm. ACM, 17, 10, p. 549-557, 1974.

3. Brinch Hansen, P., Operating System Principles, Prentice Hall, 1973.

4. Hoare, C. A. R., Communicating Sequential Processes, Comm. ACM, 21, 8, p. 666-677, 1978.

5. Brinch Hansen, P., Distributed Processes: A Concurrent Programming Concept, Comm. ACM, 21, 11, p. 934-941, 1978.

6. Dagless, E. L., A Multimicroprocessor - CYBA-M, Information Processing 77 IFIP, p. 843-848, Ed. B. Gilchrist, North-Holland, 1977.

7. Dowsing, R., Software for CYBA-M, Microprocessors and Microsystems, 3, 7, p. 306-310, 1979.

8. Sloman, M. S., The CONIC Communication System for Distributed Process Control, Communications in Distributed Systems Conference, Berlin, Jan. 1983, Springer-Verlag.

9. Dowson, M. et al, the DEMOS Multiple Processor, Euro IFIP 79, Ed. P. A. Samet, North Holland, 1979.

10. Brinch Hansen, P., The Programming Language Concurrent Pascal, IEEE Trans. Soft. Eng., 1, 2, p. 199-207, 1975.

11. Grimsdale, R. L., Halsall, F., Martin-Polo, F., and Shoja, G. C., POLYPROC II - The University of Sussex Multiple Microprocessor, Proc. IEEE 2nd. Int. Conf. on Distributed Computing Systems, p. 95-104, 1981.

12. Grimsdale, R. L., Halsall, F., Martin-Polo, F. and Shoja, G. C., MARTLET: A Programming Language for a Distributed Multiple Microprocessor System, Proc. ICS, 6th Euro. Reg. Conf. on Systems Architecture, p. 403-414, 1981.

13. Ledgard, H., ADA - An Introduction and Ada Reference Manual, Springer-Verlag, 1981.

14. Grimsdale, R. L., Halsall, F., Martin-Polo, F. and Wong, S. A., Structure and Tasking Features of the Programming Language Martlet, Computers and Digital Techniques, Proc. IEE, 129, 2, Pt. E., p. 63-69, 1982.

15. Shoja, G. C., Halsall, F., and Grimsdale, R. L., A Control Kernel to Support Ada Intertask Communication on a Distributed Multiprocessor Computer System, Software and Microsystems, 1, 5, p. 128-134, 1982.

16. Dowson, M., Collins, B. and McBridge, B., Software Strategy for Multiprocessors, Microprocessors and Microsystems, 3, p. 263-266, 1979.

17. Shoja, G. C., Halsall, F. and Grimsdale, R. L., Some Experiences of Implementing the Ada Concurrency Facilities on a Distributed Multiprocessor Computer System, Software and Microsystems, 1, 6, p. 147-152, 1982.

CLOSELY COUPLED SYSTEMS

CLOSELY COUPLED SYSTEM:

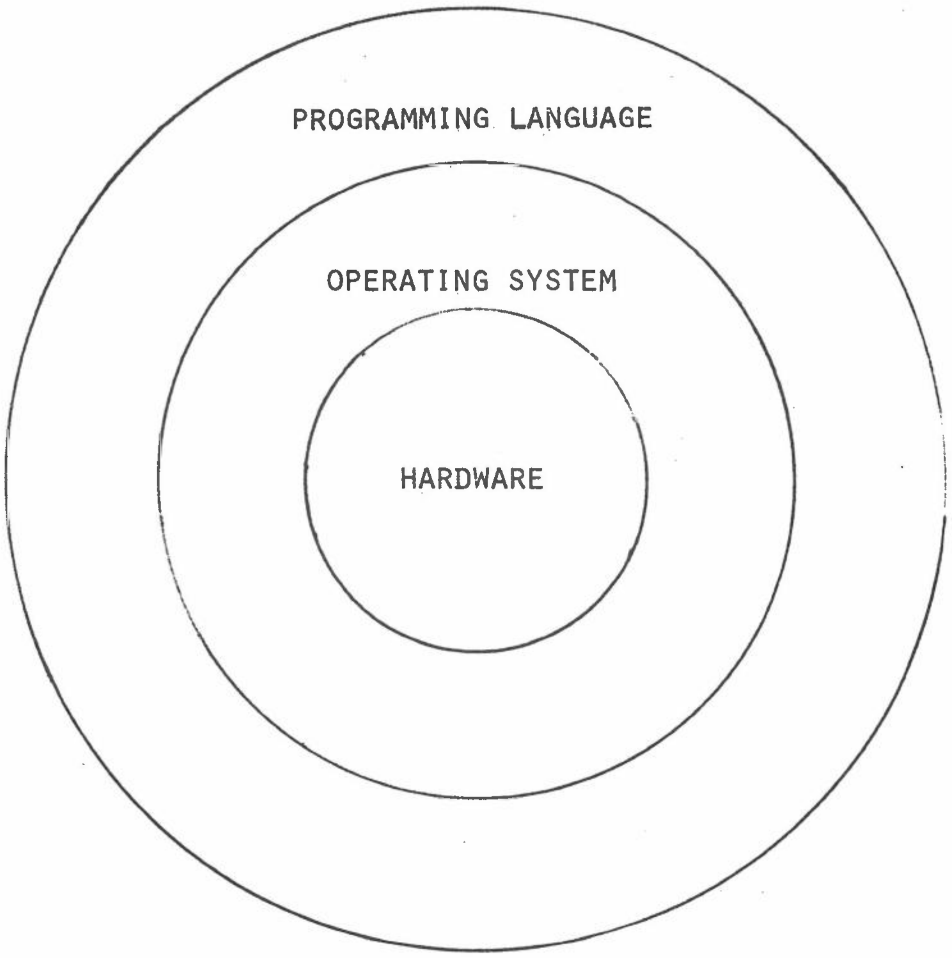
- COLLECTION OF LINKED PROCESSOR AND MEMORY MODULES
- COLLABORATIVELY EXECUTE A SINGLE JOB
- UNDER ONE MANAGEMENT

LOOSELY COUPLED SYSTEM:

- SET OF LINKED COMPUTERS
- EACH UNDER CONTROL OF SEPARATE USER
- SEND MESSAGES TO EACH OTHER
- ACCESS COMMON DATABASE

CLOSELY COUPLED SYSTEM

- SINGLE ACTIVITY
  - SUB-DIVIDED INTO COLLECTION OF TASKS WHICH EXECUTE IN PARALLEL
  - WELL-DISCIPLINED INTER-TASK COMMUNICATION


PROGRAMMING LANGUAGE
OPERATING SYSTEM
HARDWARE

EMBEDDED SYSTEMS

- PART OF A LARGER ENGINEERING SYSTEM

  INDUSTRIAL PROCESS CONTROL
  AIR TRAFFIC CONTROL
  TELEPHONE EXCHANGE CONTROL

MOTIVATIONS FOR CLOSELY COUPLED SYSTEMS

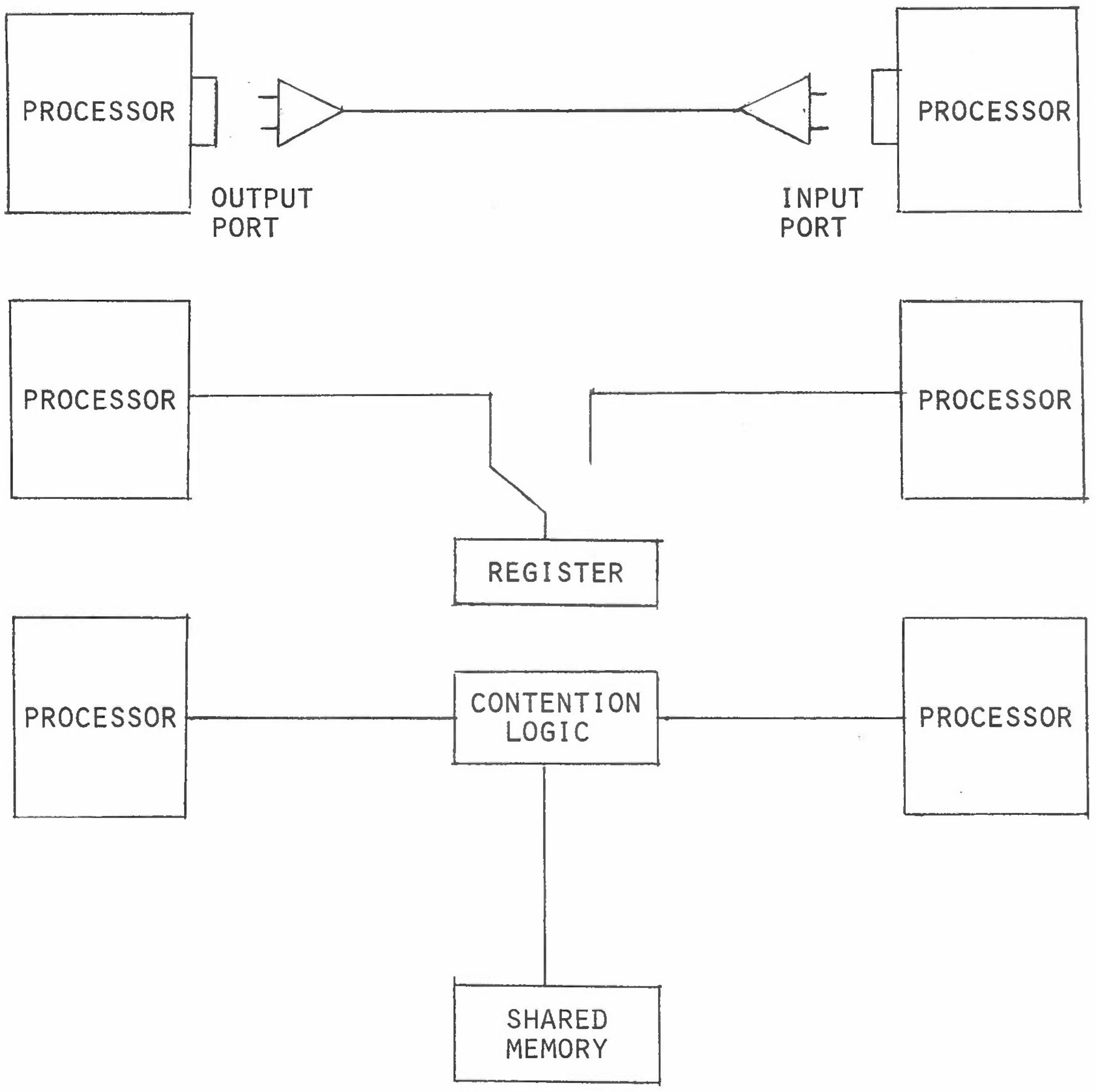
INCREASE IN SPEED OVER SINGLE PROCESSOR

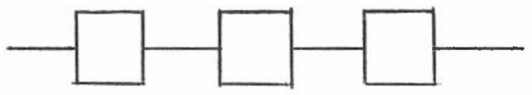
- IF SYSTEM ALGORITHM CAN BE DECOMPOSED INTO TASKS WHICH OPERATE IN PARALLEL

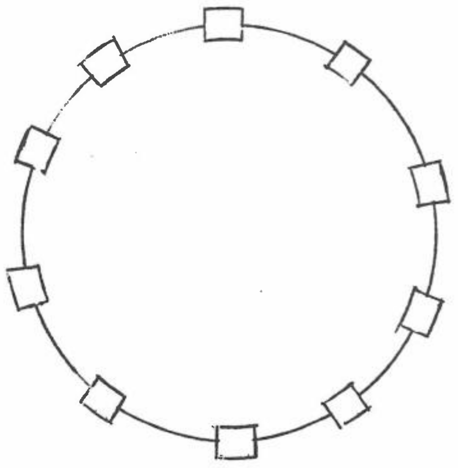
IMPROVED RELIABILITY OR AVAILABILITY

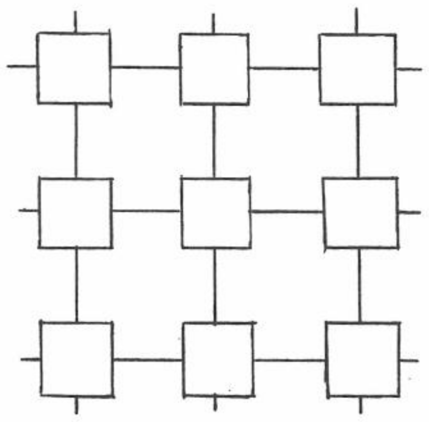
SYSTEM IS ITSELF INHERENTLY DISTRIBUTED

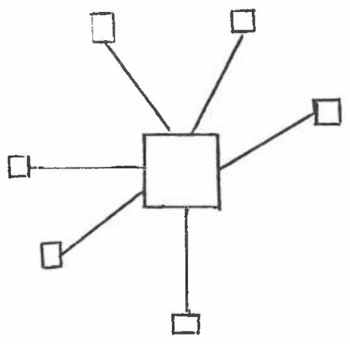
INTERCONNECTION

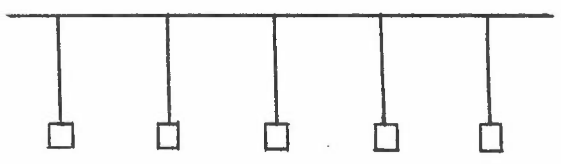




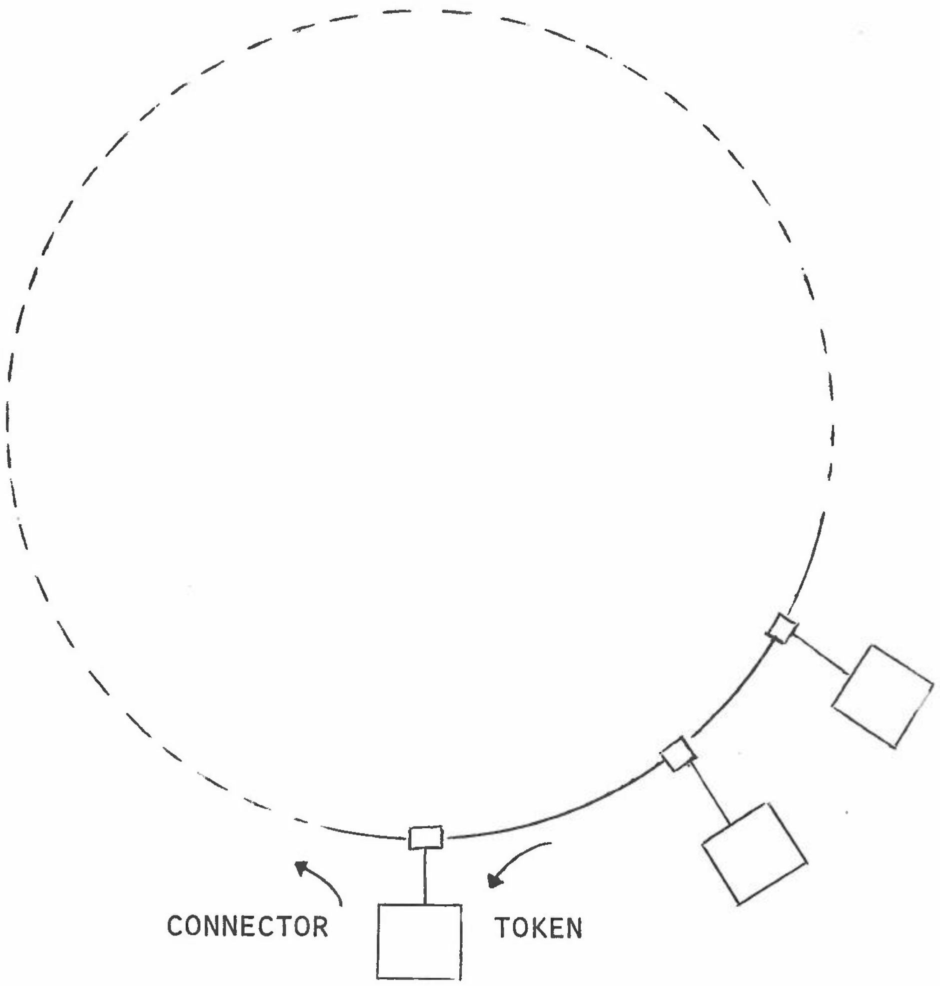




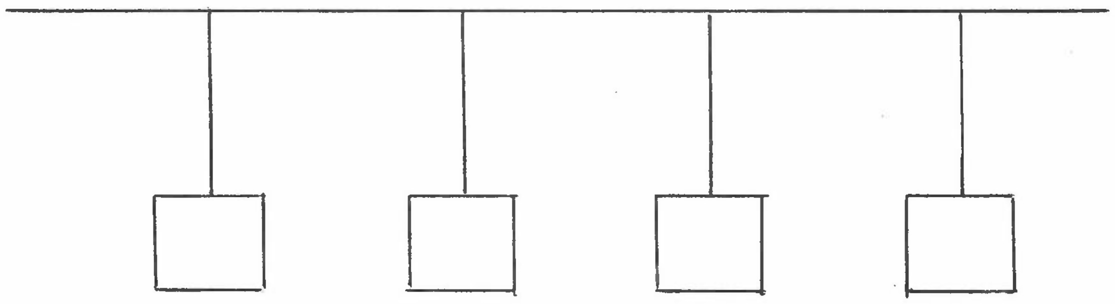




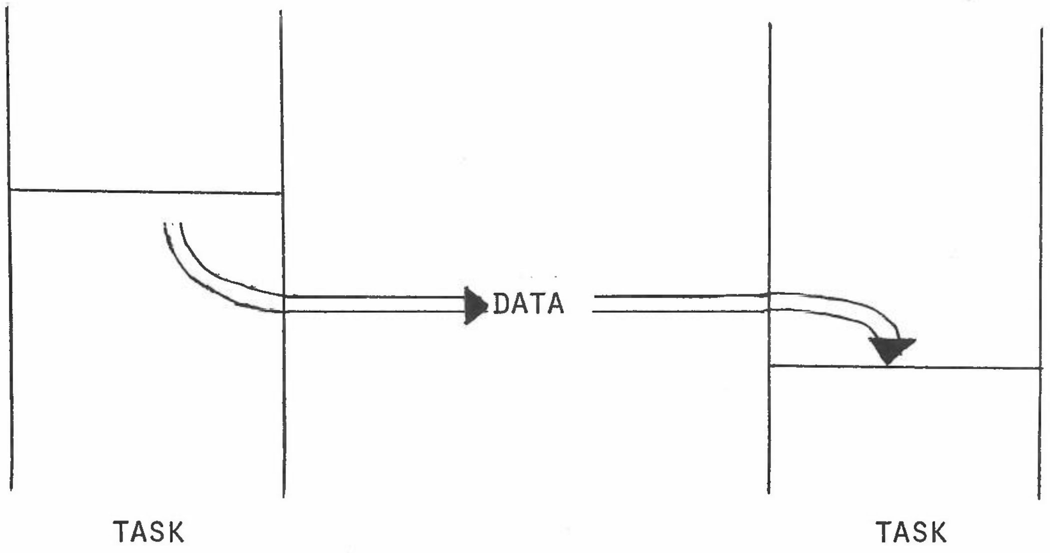
<u>TOKEN RING</u>

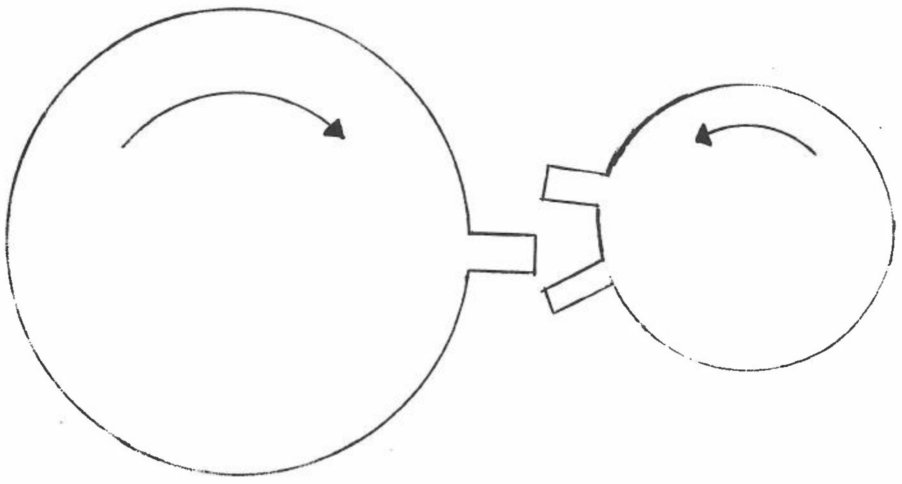


CSMA/CD

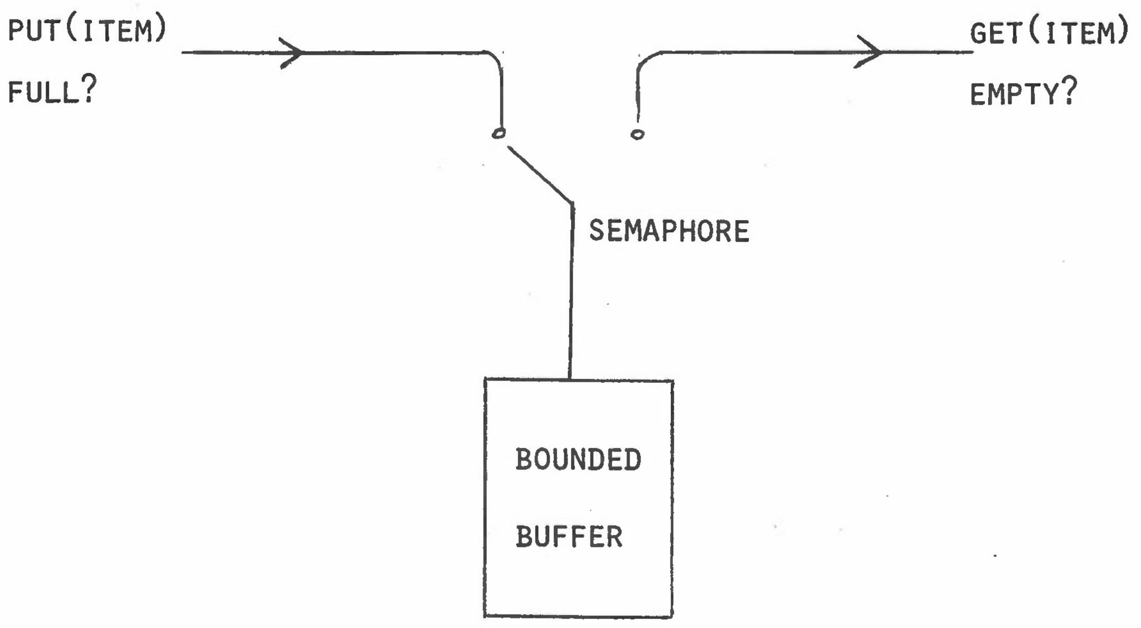


CONCURRENCY SUPPORT MECHANISMS



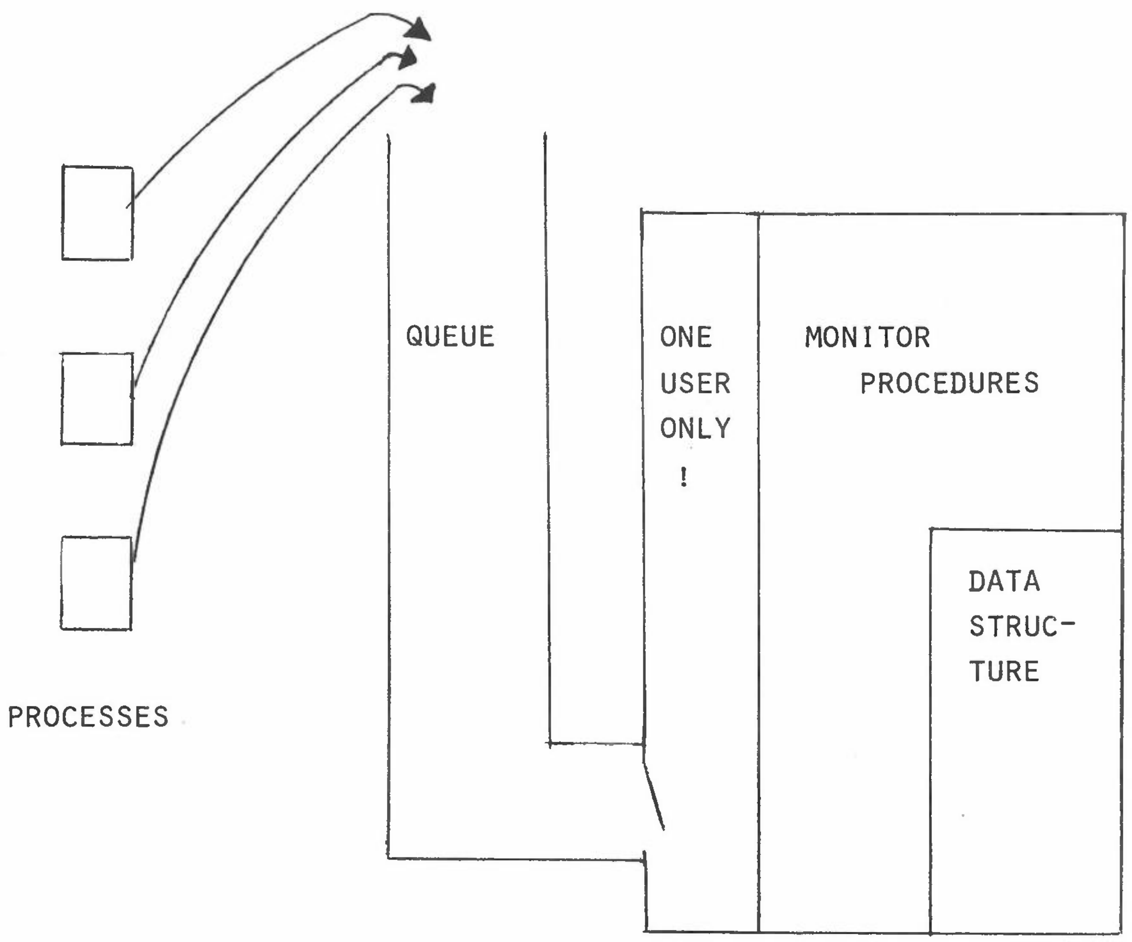


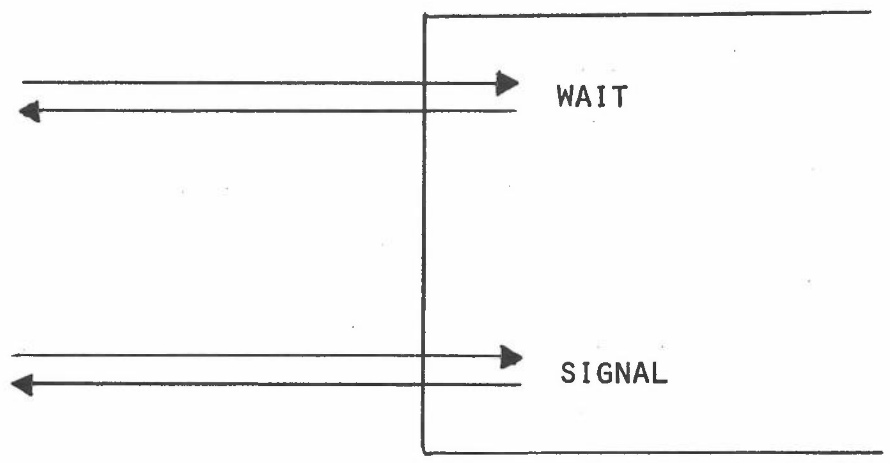
SYNCHRONISING



```
ACQUIRE SEMAPHORE                                   ACQUIRE SEMAPHORE
IF NOT FULL                                                IF NOT EMPTY
PUT(ITEM)                                                      GET(ITEM)
RELEASE SEMAPHORE                                   RELEASE SEMAPHORE
```

THE MONITOR


PROCESSES
QUEUE
ONE
USER
ONLY
!
MONITOR
PROCEDURES
DATA
STRUC-
TURE

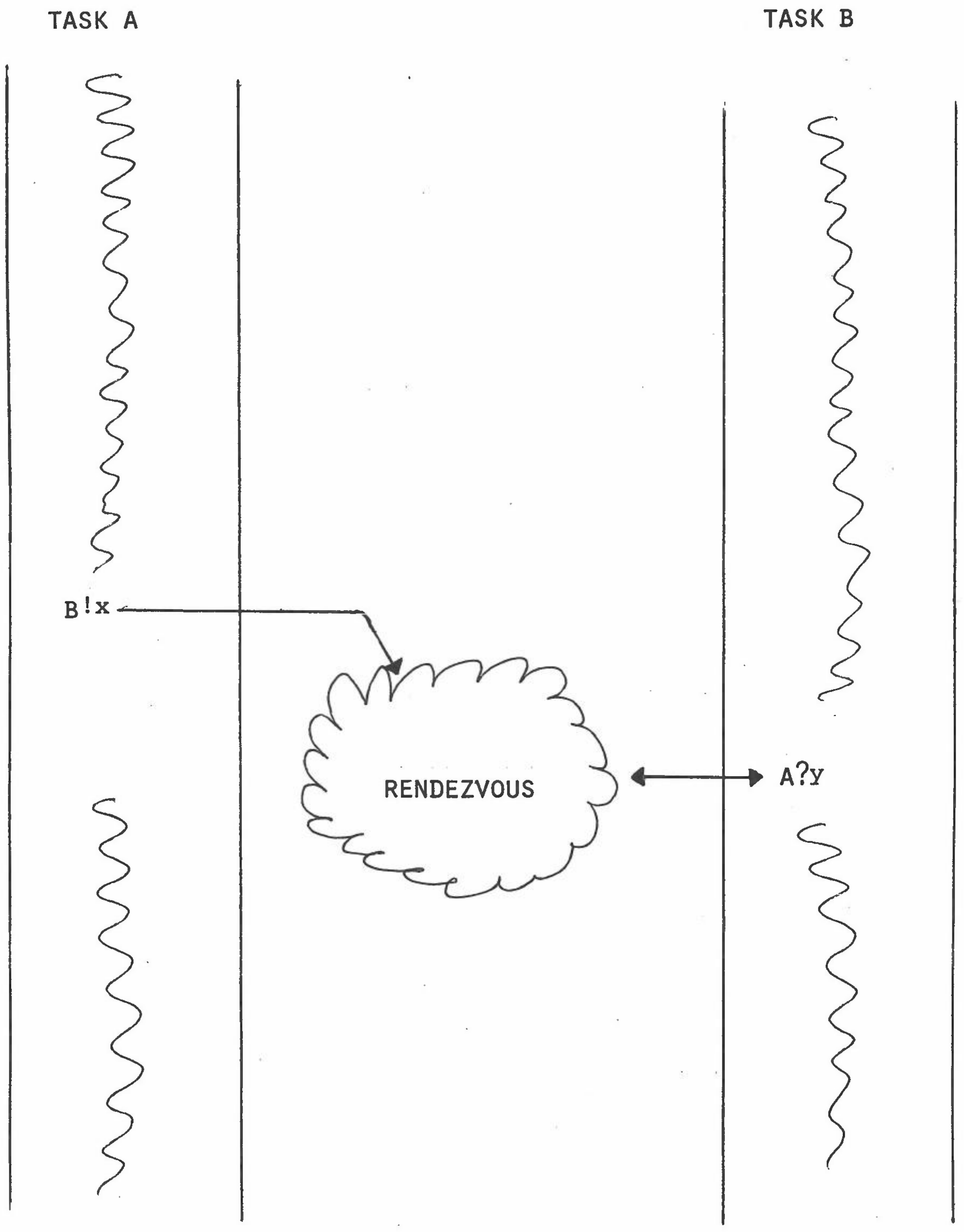

WAIT
SIGNAL

MONITOR

- ELEGANT CONCEPT
- CAN ENSURE STATES OF CALLING PROCESSES ARE DETERMINISTIC

$$\begin{tikzcd}
\text{MONITOR} \arrow[r, shift left] & \text{SUITABLE} \arrow[l, shift left] \arrow[r, shift left] & \text{SHARED MEMORY} \arrow[l, shift left]
\end{tikzcd}$$

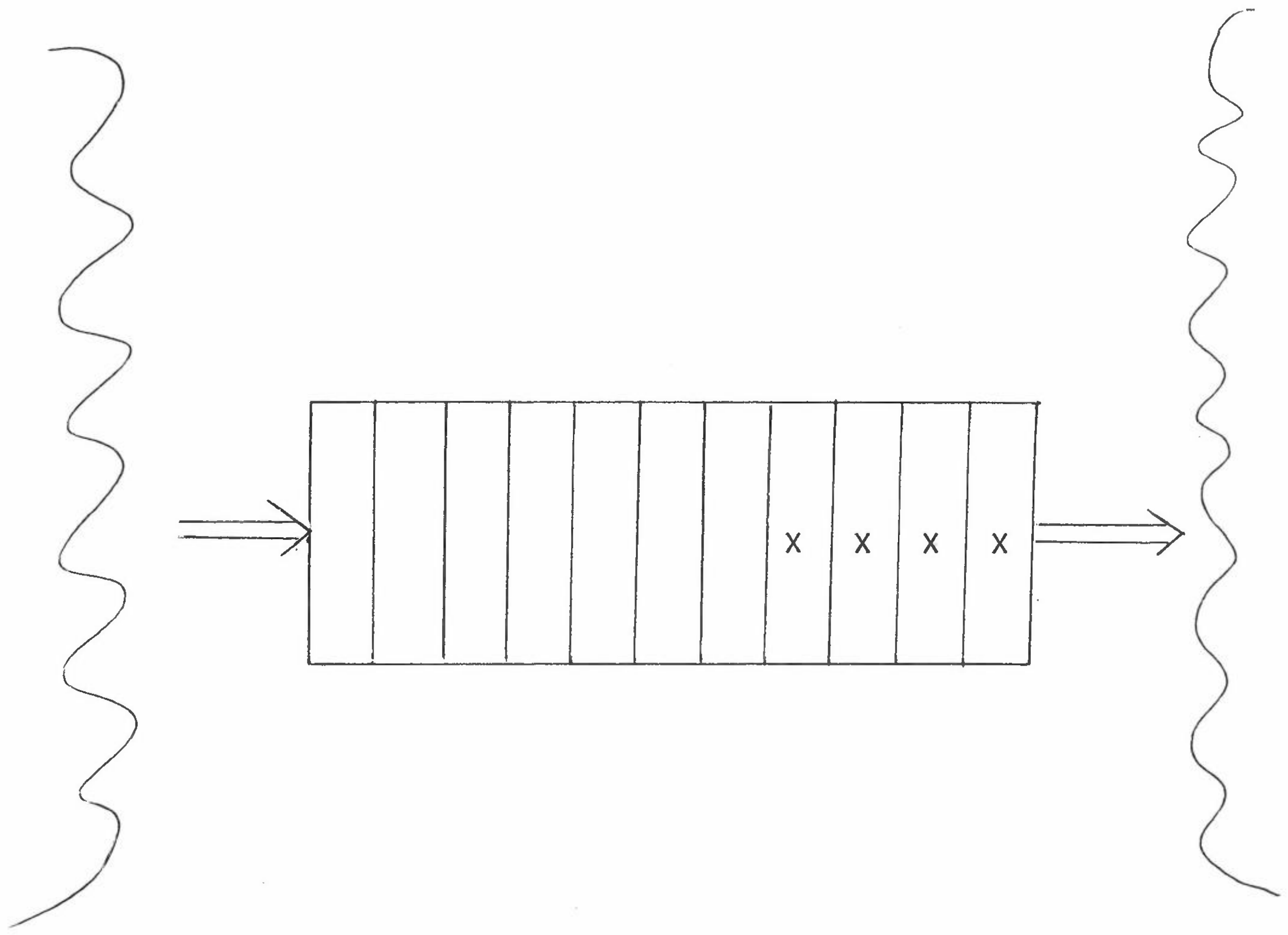
SYMMETRIC RENDEZVOUS



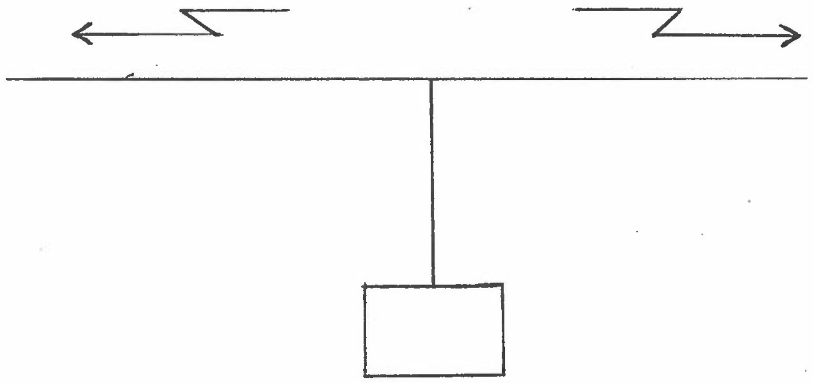
```
TASK  Z                  TASK X IS
    |                        ENTRY REQUEST;
    |                    END X;
    |
    |
    |                    TASK BODY X IS
X.REQUEST                BEGIN
    |                        LOOP
    |                            ACCEPT REQUEST
    |                            /* STATEMENTS WHICH
    |                            SERVICE REQUEST */
    |                        END LOOP;
                         END X;
```

```
SELECT
      WHEN COUNT <  MAXSIZE =>
           ACCEPT SEND_CHAR (C: IN CHARACTER) DO
                  ....
                  ....
           END;

OR
  WHEN COUNT >  0 =>
       ACCEPT RECEIVE_CHAR (C: OUT CHARACTER) DO
              ....
              ....
       END;

END SELECT;
```

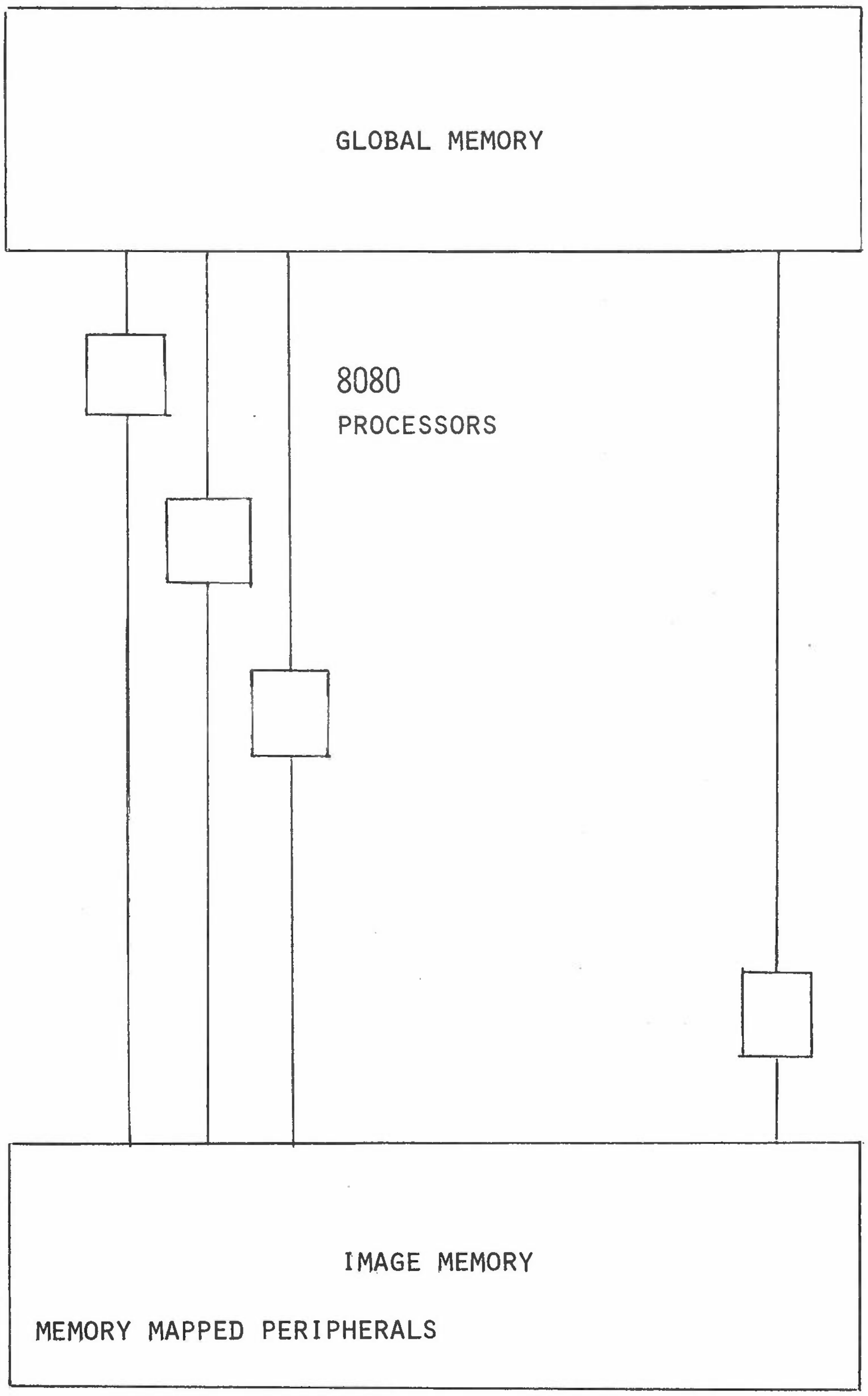
MESSAGE PASSING

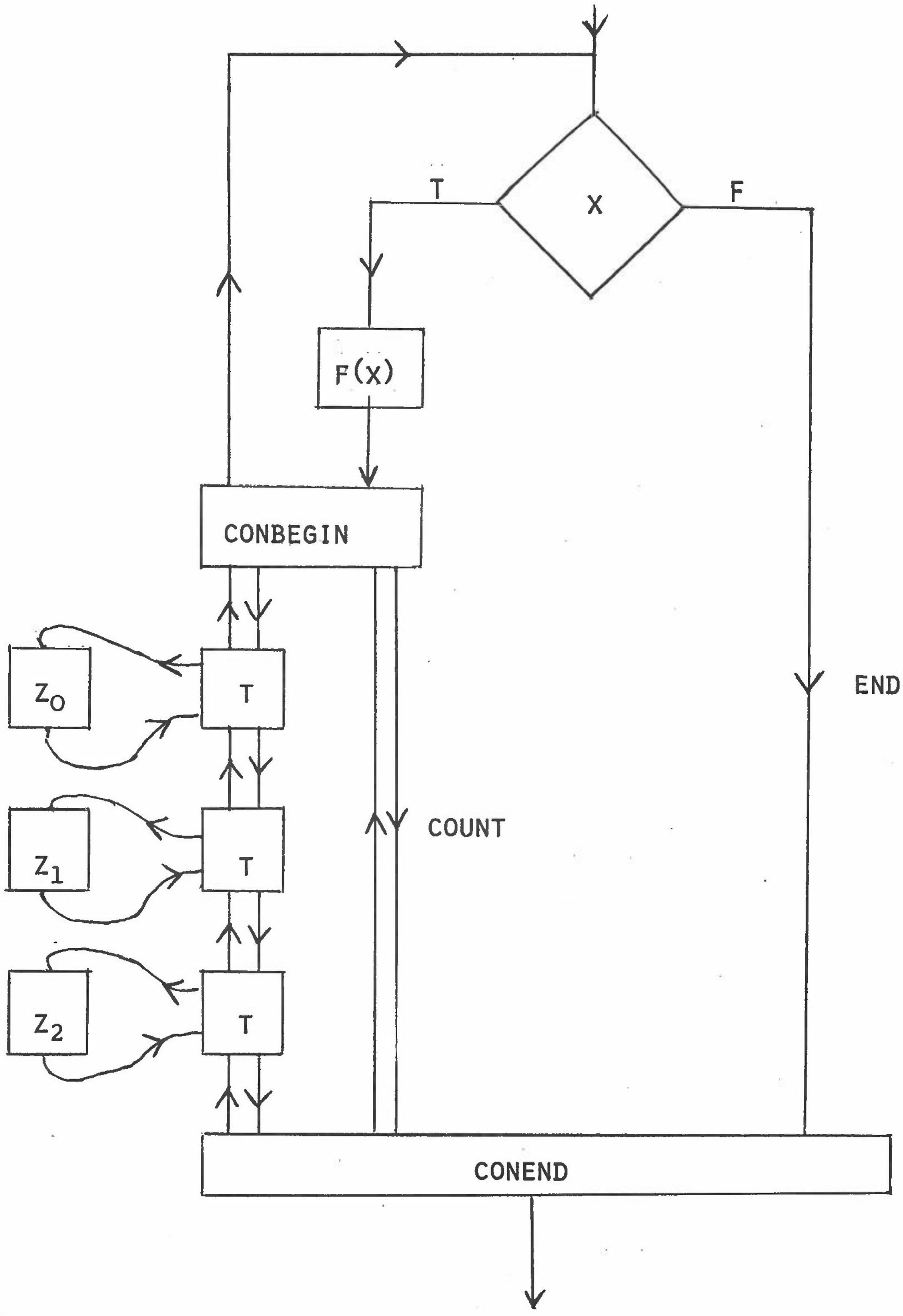


BROADCAST



CYBA-M ARCHITECTURE



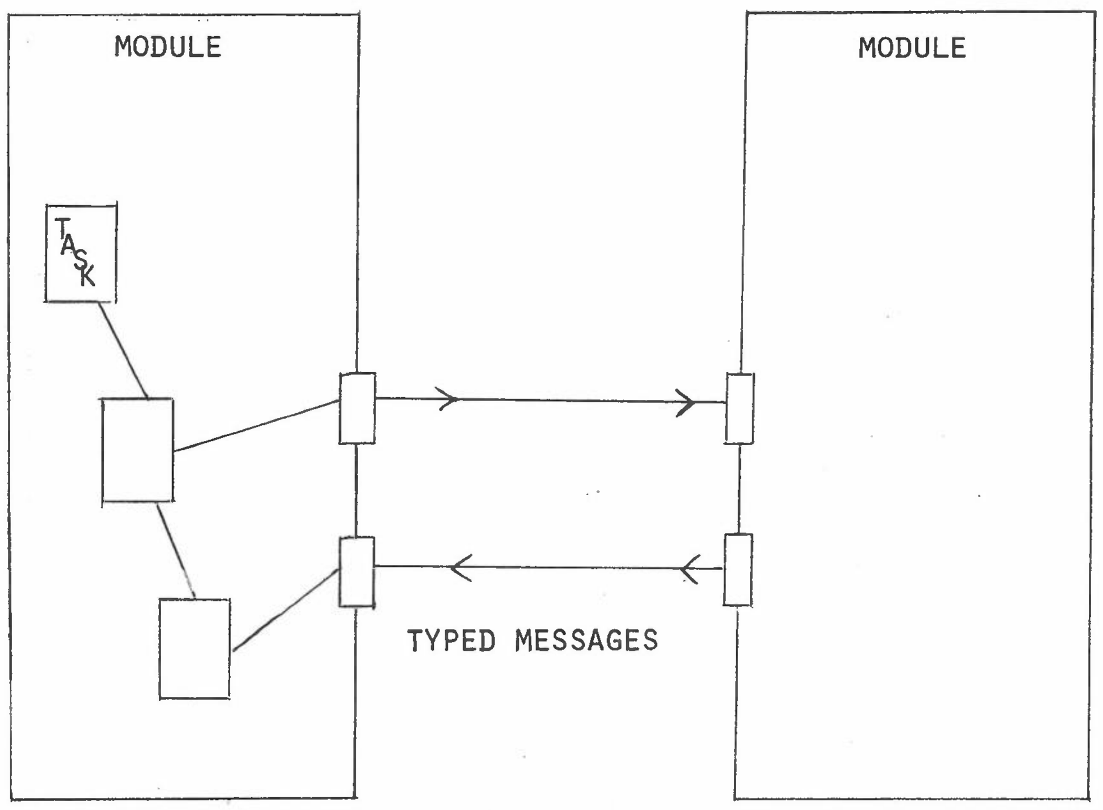


```
WHILE X
 DO  F(X)
     CONBEGIN
       Z_0; Z_1; Z_2
     CONEND
ENDWHILE
```

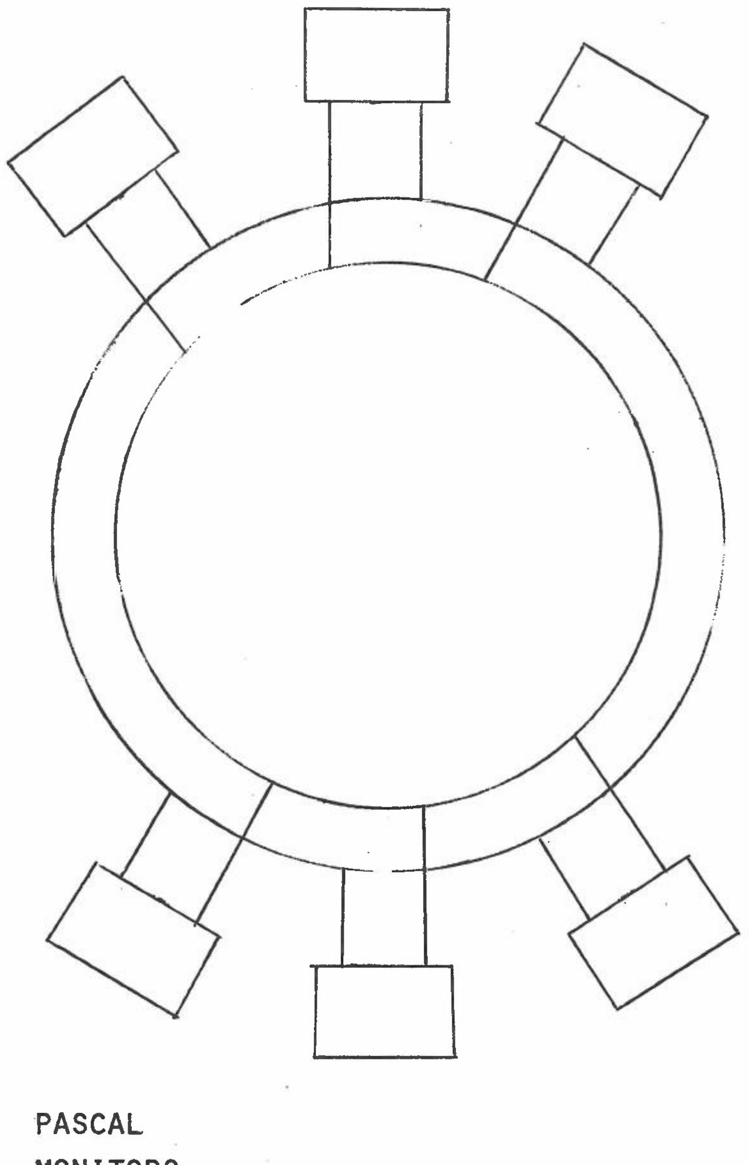
PIPELINE STRUCTURE OF A WHILE .. DO LOOP.

CONIC ICST

INTEGRATED SET OF TOOLS AND TECHNIQUES FOR CONSTRUCTING AND MANAGING LARGE DISTRIBUTED COMPUTING SYSTEMS



DEMOS   NPL AND SCICON



16 BIT
PARALLEL RING
10 MHZ CLOCK

CONCURRENT PASCAL
            MONITORS
            INTER-PROCESS REQUESTS QUEUED AT SOURCE
            3-WAY HANDSHAKE

POLYPROC UNIVERSITY OF SUSSEX

FOR EMBEDDED SYSTEMS

MARTLET PROGRAMMING LANGUAGE

- SEQUENTIAL PASCAL
- INTER-TASKING LIKE ADA
    - TASK MODULES
    - ENTRY-CALL/ACCEPT
    - SELECT
    - INSTANCING
    - EXCEPTION HANDLING

APPLICATIONS PROGRAMMER(S) CREATE SUITE OF TASK MODULES WITHOUT THE NEED TO KNOW ABOUT CONFIGURATION OF TARGET ARCHITECTURE

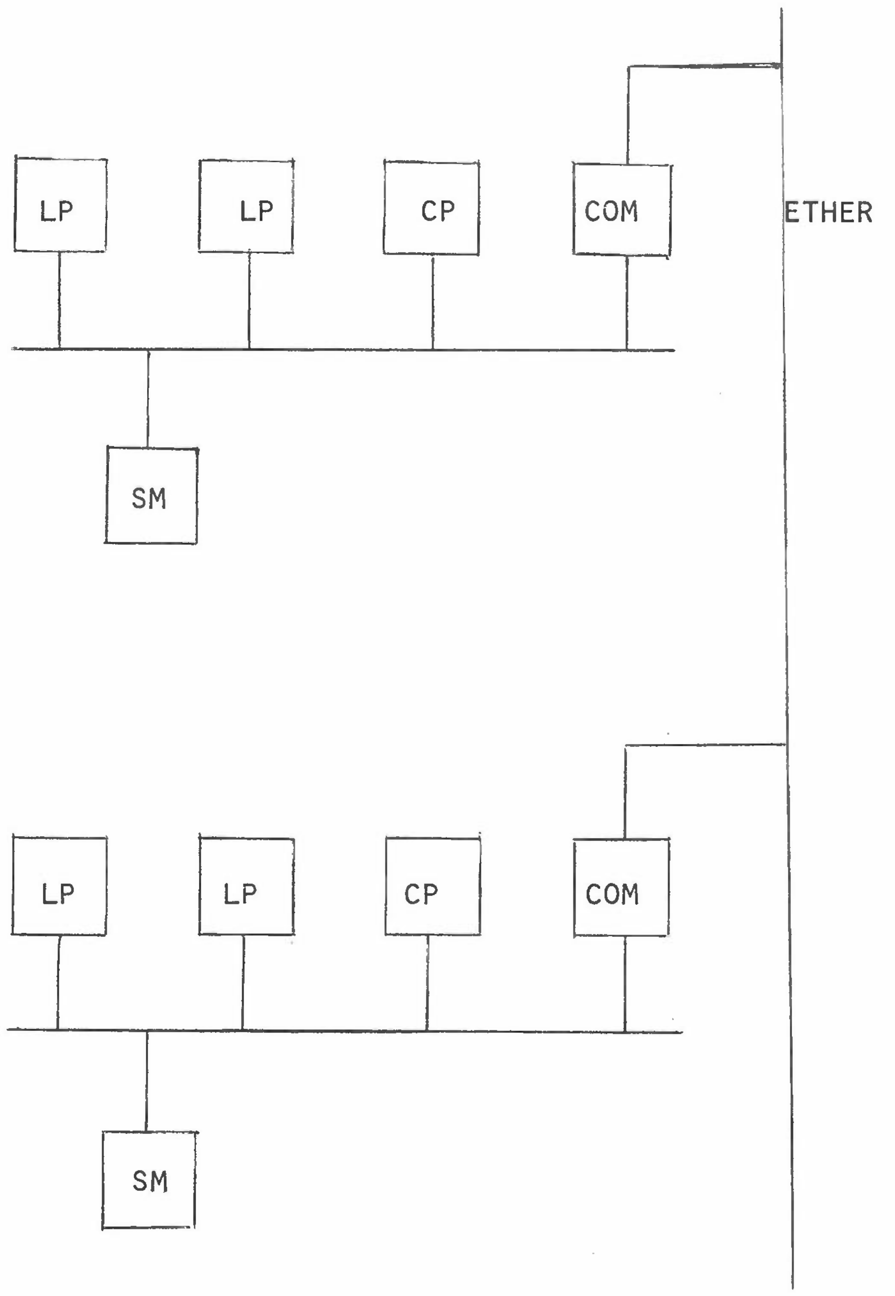
TASK MODULES CAN BE SEPARATELY COMPILED
STATIC ASSIGNMENT AT SYSTEM CONFIGURATION PHASE

COMPILER AND CONFIGURATION PROGRAM ARE BOTH WRITTEN IN PASCAL

POLYPROC HARDWARE CONFIGURATION

NOT CONSTRAINED TO ONE TYPE OF MICROPROCESSOR

SEVERAL TASKS IN ONE STATION



INTEL 86/12A
MULTIBUS
OWN DEISIGN-
COMMUNICATIONS
PROCESSOR

CP	CONTROL PROCESSOR - MANAGES INTER-TASKING AND TASK SCHEDULING IN PARALLEL WITH TASK EXECUTION ON LOCAL PROCESSORS
COM	COMMUNICATIONS PROCESSOR - SUPPORTS CSMA/CD PROTOCOL
LP	LOCAL PROCESSORS - EXECUTE APPLICATION TASKS
SM	SHARED MEMORY - SUPPORTS INTER-TASK COMMUNICATION BETWEEN TASKS IN A STATION

POLYPROC RUN-TIME SUPPORT SOFTWARE

MARTLER COMPILER GENERATES M-CODE - EXECUTED INTERPRETIVELY

    SLOWER EXECUTION
    PORTABLE
    INTERMEDIATE CODE IS COMPACT
    CAN INCORPORATE PERFORMANCE
        MEASUREMENT TECHNIQUES

CONTROL KERNEL IN CONTROL PROCESSOR

    SUPPORTS INTER-TASKING
    AND TASK SCHEDULING

SMALL LOCAL KERNEL IN LOCAL PROCESSOR - FOR TASK/CONTEXT SWITCHING

CONTROL KERNEL

- TRANSFORMS SYSTEM INTO VIRTUAL MACHINE SUPPORTING CONCURRENT FEATURES OF LANGUAGE
- PROVIDES INTER-TASKING WITHIN STATION
- TRANSPORT TASKS PROVIDE INTER-STATION COMMUNICATION
- SOLE OWNER OF ALL VITAL AND CRITICAL DATA STRUCTURES FOR STATION OPERATION

CONTROL KERNEL DATA STRUCTURES

SUITE OF TASK ACTIVATION RECORDS (ONE PER TASK MODULE RESIDENT IN STATION)

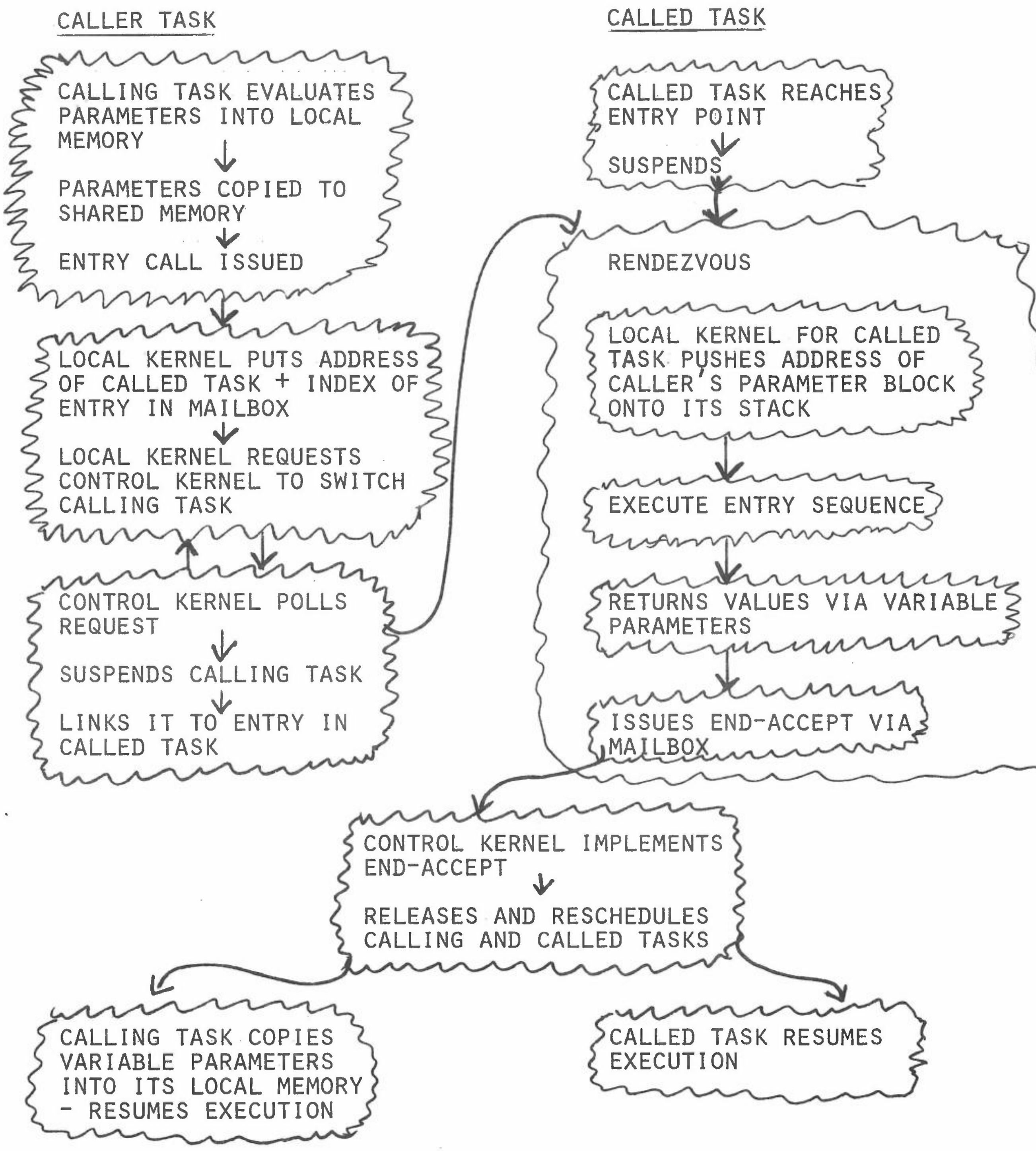
SET OF PROCESSOR STATUS RECORDS (ONE PER LOCAL PROCESSOR)

STATION DIRECTORY - NAMES AND LOCATIONS OF EACH TASK

INTERRUPT MAP TABLE - TO MAP INTERRUPTS TO SERVICE TASKS

SET OF FLAGS - INTERRUPT AND COMMUNICATION CHANNELS

IMPLEMENTATION OF RENDEZVOUS MECHANISM



CALLER TASK
CALLED TASK
CALLING TASK EVALUATES PARAMETERS INTO LOCAL MEMORY
PARAMETERS COPIED TO SHARED MEMORY
ENTRY CALL ISSUED
LOCAL KERNEL PUTS ADDRESS OF CALLED TASK + INDEX OF ENTRY IN MAILBOX
LOCAL KERNEL REQUESTS CONTROL KERNEL TO SWITCH CALLING TASK
CONTROL KERNEL POLLS REQUEST
SUSPENDS CALLING TASK
LINKS IT TO ENTRY IN CALLED TASK
CALLED TASK REACHES ENTRY POINT
SUSPENDS
RENDEZVOUS
LOCAL KERNEL FOR CALLED TASK PUSHES ADDRESS OF CALLER'S PARAMETER BLOCK ONTO ITS STACK
EXECUTE ENTRY SEQUENCE
RETURNS VALUES VIA VARIABLE PARAMETERS
ISSUES END-ACCEPT VIA MAILBOX
CONTROL KERNEL IMPLEMENTS END-ACCEPT
RELEASES AND RESCHEDULES CALLING AND CALLED TASKS
CALLING TASK COPIES VARIABLE PARAMETERS INTO ITS LOCAL MEMORY - RESUMES EXECUTION
CALLED TASK RESUMES EXECUTION

CALLING AND CALLED TASKS IN DIFFERENT STATIONS

DYNAMIC APPROACH

REQUIRES MEMORY ALLOCATION FOR PARAMETER BLOCK
- BUT THIS IS NOT PRE-ALLOCATED

MEMORY ALLOCATED WHEN REQUIRED AT RECEIVING STATION

MESSAGES (PARAMETER BLOCKS) QUEUED AT SENDER

TRANSMISSION CONTROLLED BY THREE-WAY HANDSHAKE PROTOCOL

CALLING AND CALLED TASKS IN DIFFERENT STATIONS

STATIC APPROACH

(USED IN POLYPROC)

PAIR OF INTERMEDIATE TRANSPORT TASKS ARE INTRODUCED AUTOMATICALLY AT SYSTEM CONFIGURATION TIME

OPERATION IS TRANSPARENT TO CONTROL KERNELS
- APPEARS THE SAME AS INTRA-STATION COMMUNICATION

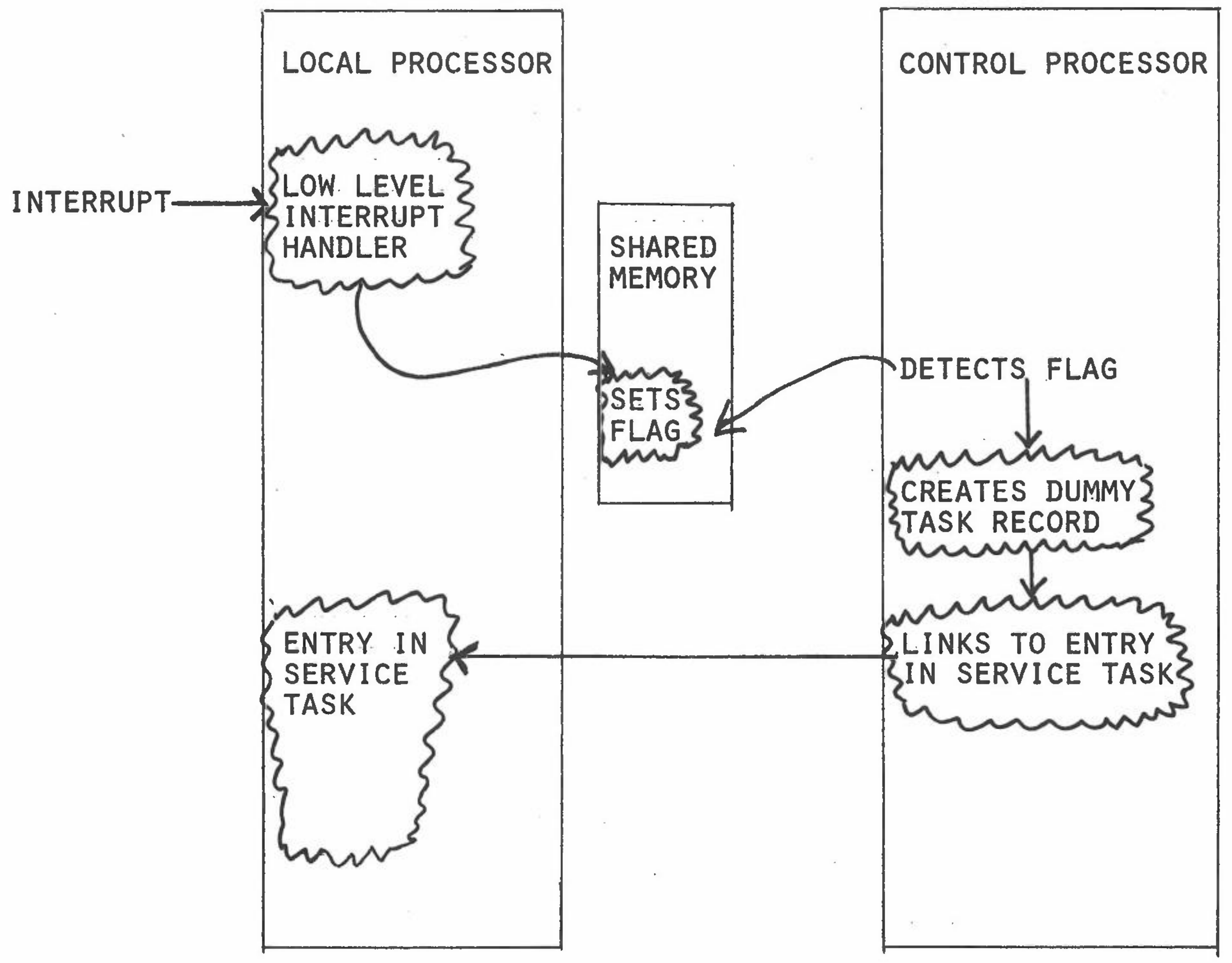
THIS FUNCTIONAL SEPARATION PERMITS A DIFFERENT LOW-LEVEL COMMUNICATION LAYER TO BE SUBSTITUTED IF DESIRED

GREATER OVERHEADS BETWEEN STATIONS COMPARED WITH WITHIN STATIONS

INTERRUPT HANDLING

DESIRABLE TO HANDLE INTERRUPTS IN A WAY WHICH IS STRUCTURED AND INDEPENDENT OF HARDWARE

INTERRUPT IS REGARDED AS AN ENTRY TO A TASK WHICH INCLUDES AN ACCEPT STATEMENT FOR THAT INTERRUPT



The New Programming: Functional and Logic Languages

John Darlington
Department of Computing
Imperial College, London

Introduction

Although there are a bewildering variety of computers currently available together with an equally bewildering variety of languages with which to program them, there is a real sense in which all machines and languages in use today are the same. All computers in common use are based on the von Neumann principle around which the very first succesful computers were built. They thus all operate by having a program stored in memory through which control flows causing operations to be performed reading and altering memory locations. At any one time one instruction in the program is being obeyed, thus the machines are inherently sequential. Almost all the programming languages in use today faithfully follow this model. Thus even so called high level languages such as Pascal have sequential control flow and assignment as their fundamental characteristics.

Certain languages designed primarily for systems work, such as Ada, extend this model somewhat by incorporating tasking facilities that allow control to be at several points simultaneously allowing distributed systems to be modelled. However these facilities are built on top of a sequential language and crucially all concurrent activity must be anticipated and controlled by the programmer. Although one can conceive of systems of, say, up to ten separate processes being explicitly programmed, systems of a million processes are beyond human comprehension and control.

This, almost total, reliance on von Neumann machines and languages has led computing into some critical dead ends. On the software side despite valiant efforts and many innovations we are still incapable of developing and maintaining even moderately complex systems with the degree of reliability and precision that should characterise an engineering or scientific discipline.

Considering the great strides computers have had in increasing productivity in many diverse areas of commerce and industry it is ironic that an activity central to their use, programming, has remained largely unmechanised. Of course sophisticated editors and efficient compilers have brought about significant increases in programmer productivity, but the central intellectual task of programming, that of devising the algorithm to be employed to meet the stated requirements, is still very much an art and incapable of being mechanised to any significant degree. It is in this area (and the related one of program modification and maintenance) that mostof the cost and unreliability of software development is to be found. What we desperately need to bring the power of computers to bear on this problem is ways of dealing with specifications, programs and the relationship between them at levels above that of simple text. It has been found impossible to develop such a calculus of program development using the conventional languages.

On the hardware side incredible amounts of effort and expense have been and are continuing to be devoted to attempting to get machines to run faster. By increasing switching speeds, shrinking connection distances and improving internal organisation increases in performance can still be won. But the sequential, one thing at a time, restriction places fundamental limitations on how far this process can be carried. The obvious answer is to go to parallel evaluation and have many things going on at once. However if we attmpt to do this while still following the basic von Neuman model and more importantly attempting to execute von Neumann based languages that are inherently sequential we run into severe difficulties. Extensive co-ordination and communication is needed between processors to ensure that different parts of the program are executed in the sequence prescribed and this soon outweighs any gains achieved. Thus projects aimed at building multi-processor machines for conventional languages have by and large been failures apart from specialised applications such as vector or array processors.

In contrast the declarative languages have very different origins. They are based on various mathematical formalisms developed initially in the study of formal deduction and computation. The two main classes of these languages are the logic programming languages based on the first order predicate calculus and the functional programming languages based on the lambda calculus and recursion equations. This pedigree gives the declarative languages fundamentally different properties from the conventional ones. Being faithfully derived from mathematical formalisms they inherit all the desirable, indeed essential, properties of such notations that give mathematics its unique power. Conventional languages most certainly do not have these properties. The most important of these properties goes by the technical name referential transparency and requires that equivalent expressions can be substituted freely for each other without altering he meaning of the expression they are substituted into. For example arithmetic is referentially transparent so 5 can be freely substituted for the expression 3 + 2. Another way of saying this is that the meaning of an expression should depend solely on its textual context and not at all on any notion of the history of the computation being performed.

This, it is claimed, conveys fundamental advantages on the declarative languages and enables them to break the log jams at present being encountered in computing. Firstly these languages are intrinsically more powerful, concise and understandable programming vehicles for the reasons alluded to above. Furthermore because of their simple substitutive properties formal manipulation of programs becomes feasible and we can develop simple calculi for conducting program developments in, guaranteed to preserve the meaning of the programs thus manipulated. The development of such calculi seems an essential requirement if programming is ever to become a science (and therfore mechanisable) rather than a craft.

Secondly because the meaning of any subpart of a declarative language program is independent of the meaning of other textually disjoint subparts the meaning of the whole is independent of the order of the evaluation of the parts and they can therefore be evaluated in parallel. Thus parallel evaluation of these languages is the natural model of evaluation and can proceed with a minimum of co-ordination or communication. This form of parallel evaluation is therefore random or natural and need not be programmed in explicitly. Thus the degree of concurrency exploitable is limited only by the degree of parallelism in the algorithm employed and the number of processing units available. As algorithms can be easily written that, in declarative languages, give rise to exponential amounts of parallelism it can easily be seen that the speed of computers based on declarative languages is no longer limited by the technology employed but by the number of processors available. This is a much healthier position to be in especially as the new VLSI technology dictates that machines can be much more economically built out of many identical replicated components rather than a few complex ones.

In section 1 we introduce functional and logic programming languages and give simple examples of their use. Section 2 lists some existing declarative languages and their implementations while section 3 outlines some of the main applications the languages have so far been put to. Sections 4 and 5 very briefly introduce the ideas of program transformation and parallel evaluation that make declarative languages so attractive. Section 6 discusses some issues involving the declarative languages and problems that need to be overcome, while section 7 speculates on the future for the languages.

1. Declarative Languages

In this section we will attempt very briefly, to introduce the functional and logic programming languages. Fortunately the language's simplicity aids this task.

1.1 Functional Languages

Programs in a functional language consist simply of sets of equations defining functions in terms of other simpler or primitive functions.

For example

```
     max(x,y) = if x > y then x else y
```

employs the primitive *if then else* to define a function, max, to calculate the maximum of two numbers.

The program

```
     maxof3(x, y, z) = max(max(x,y), z)
     max(x,y) = if x > y then x else y
```

defines two functions max as above and maxof3 to calculate the maximum of three numbers. Note that there is no ordering implied on the equations.

Functions can be defined in terms of themselves,

```
     factorial(n) = if n = 0 then 1 else n * factorial(n-1)
```

defines the familiar factorial function.

Several functional languages instead of having just one equation for each function, have separate equations, one for each case of the input parameter, thus

```
     factorial(0) = 1
     factorial(n+1) = n+1 * factorial(n)
```

defines the same factorial function. The cases covered by each equation must be disjoint.

Executing a functional program is very simple. It consists of taking an expression and rewriting it according to the equations of the program until no further rewritings are possible. The equations are thus used in a left to right manner as reduction rules. For example, to compute the factorial of 5 given either of the above programs our starting expression would be factorial(5) and the reductions would proceed factorial(5) => 5 * factorial(4) => 5 * (4 * factorial(3))

=> 5 * (4 * (3 * factorial(2))) => 5 * (4 * (3 * (2 * factorial(1))))

=> 5 * (4 * (3 * (2 * (1 * factorial(0))))) => 5 * (4 * (3 * (2 * (1 * 1))))

=> 120   (Assuming * as a built in function).

Note that functional programs are **deterministic**, in that for any given program and starting expression there is only one possible answer. Moreover functional programs have the **Church-Rosser** property, that is if at any stage of a computation several equations may be applied next the final answer, if one is found, is independent of which particular equation is chosen. Thus different **modes** of evaluation can be explored without affecting the correctness of the final result.

Structures are very simply accommodated in the declarative languages by introducing functions, called constructor functions, and using expressions constructed out of these functions to name structures. Thus lists have two constructor functions **nil**, the empty list and **cons** which takes an element and a list and builds a list equivalent to the original but with the element added at the front. Thus the term cons(1, nil) names the list with one element, 1, and cons(1, cons(2, cons(3, nil))) names a three element list. Cons is often written as an infix operator, " . ". Thus 1.nil is the same as cons(1, nil).

Equations can be written over structures just as easily as over scalars

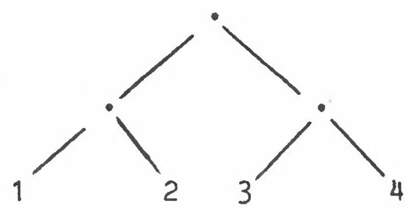
```
     length(nil) = 0

     length(x.1) = 1 + length(1)
```

Defines a function to calculate the length of a list.

This ability to introduce new structures by simply introducing new constructor functions means that what would be considered abstract data types in more conventional languages can be introduced and directly manipulated in declarative languages. Thus binary trees can be defined by introducing two new constructor functions, **atom** to construct tips of binary trees and **tree** to construct the interior nodes. Thus the term

```
     tree(tree(atom(1), atom(2)), tree(atom(3), atom(4)))
```

names the binary tree



and

```
frontier(atom(n)) = n.nil

frontier(tree(t1, t2)) = append(frontier(t1), frontier(t2))

append(nil, 1) = 1

append(x.l1, l2) = x.append(l1, l2)
```

defines a function frontier that returns a list of the elements of a tree produced by a left to right traversal. Append joins two lists together.

All functional languages in serious use today are higher order. This means that functions are treated as first class objects and can be passed as paramaters and returned as values. Only Algol-68 amongst the conventional languages approaches this simple but powerful facility.

For example a function map can be simply defined that takes a function as a paramater and applies it to every element of a list

```
map(nil,f) = nil
map(x.l,f) = f(x).map(l,f)
```

The existence of such functions makes program writing much easier. Thus a function, double, that doubles every element of a list can be written in terms of map without employing any explicit recursion

```
double(l) = map(l, times2)
times2(n) = 2 * n
```

Lambda expressions can be used to define functions locally without giving them global names. Thus lambda n. 2*n is an expression denoting a function that on being given a number will multiply it by 2. Thus double can be written

```
double(l) = map(l, lambda n. 2*n)
```

The ability to define function building functions is very powerful. For example

```
compose(f,g) = lambda n. f(g(n))
```

takes two functions and returns the function that is their composition.

The availability of higher order functions encourages very powerful programming styles. Often one approaches a problem by defining the data structures naturally involved together with a collection of higher order 'iterators' that walk over these structures applying functions given as paramaters. If this is done correctly programming the application largely consists of suitably instantiating these high level iterators and there is little or no explicit recursion in the main program with consequent gain in reliability and understandability.

The use of function forming functions, such as compose, carries this process even further, and encourages the use of 'program forming operations'. That is algorithms represented as functions can be operated on directly to produce more powerful algorithms.

1.2 Logic Programming

As functional programming can be viewed as a computational use of the lambda calculus so logic programming is a computational realisation of the first order predicate calculus. In this calculus the primitive statements concern relationships between individuals for example Father(John,Heather), Heavy(Lead) and $Times(2, 2, 4)$. Present implementations of the wider logic programming ideal are all based on the Horn cluse subset of logic. That is all statements in a logic program have the form of either atomic assertions

```
     Mother(Kate, Heather)
```

asserting the fact that Kate is the mother of Heather; or general rules

```
     Grandparent(x, y) if Parent(z, y) and Parent(x, z)
```

stating the fact that you are the grandparent of someone if there is some individual (z in the above rule) who is your child and the parent of the purported grandchild.

Thus a logic program is a collection of clauses stating either general rules or specific facts

```
     Grandparent(x, y) if Parent(z, y) and Parent(x, z)

     Parent(x, z) if Father(x, z)

     Parent(x, z) if Mother(x, z)

     Mother(Ann, Kate)

     Father(Maurice, Kate)

     Mother(Kate, Heather)
```

Running a logic program consists of querying this data base. Thus we can ask if certain relations hold given the facts contained in the program. The simplest queries are those that have yes/no answers such as

```
     Grandparent(Ann, Heather) ?
```

This simple theorem is implied by the above program regarded as a set of assumptions so the user answer to such a query would be yes.

Other ways of querying the program involve constructing answers. Thus a query of the form

```
     Grandparent(Ann, x) ?
```

where x is a variable asks the program to exhibit any x which satisfies the relation. Given the above program there is only one such individual, Heather, which will be found as a result of running the logic program.

Logic programs have a wonderful versatility in that the same program can be queried in many different ways. For example a query

```
     Grandparent(x, Heather) ?
```

asks the program to try and find the grandparents of Heather. In this case we would hope the program would come up with two answers for x, Maurice and Ann. Note that in contrast with the functional languages logic programs are **non-deterministic**, that is, there may be several possible answers for any given input.

This style of programming can equally well be applied to more conventional problems, for example

```
Factorial(0, 1)

Factorial(n+1, w) if Factorial(n, u) and Times(n+1, u, w)

Length(nil, 0)

Length(x.l, w)     if Length(l, u) and Plus(u, 1, w)
```

define the Factorial and Length relations analogous to the factorial and length functions defined earlier. Again these programs can be queried in many ways

```
Length(1.(2.nil), u) ?
```

asks the program to find the length of a given list, but

```
Length(x, 2) ?
```

asks the program to construct a list of length 2. In the second case the answer found would be u.(v.nil) where u and v are variables, an expression representing all the lists of length 2.

All current implementations of the logic programming ideal are variants of the language PROLOG. This employs the Horn clause subset of logic as described above and imposes extra restrictions to make the running of programs feasible on sequential machines. In PROLOG the rules are tried in the order they are written i.e. top down and left to right within a rule and the interpreter employs backtracking to explore alternative solutions. This and other features added to assist efficient sequential implementations mean that PROLOG is not strictly speaking a declarative language.

As well as being a uniquely powerful and flexible programming language logic has applications beyond those of conventional languages. For example the same formalism can be used as a data base model and query language, for knowledge representation and problem solving in artificial intelligence applications, and for implementing expert systems.

2. Implementations

The U.K. is extremely fortunate in having many of the pioneers and principle exponents of functional and logic programming. In this section we will very briefly mention most of the principal U.K. activity with reference to some of the more important work abroad.

2.1 Functional Languages

HOPE is a higher order recursion equation based functional language initially developed and implemented at Edinburgh, Burstall et al [1980]. HOPE has strong polymorphic typing. This conveys all the advantages of strong typing with a much greater degree of flexibility. HOPE was initially implemented in POP-2 on the SERC's DEC-10 at Edinburgh. Our own group at Imperial is now making extensive use of HOPE and continuing its development. We have developed a Pascal interpreter for HOPE that has been used very successfully for teaching and transported to several machines including a PERQ. Our parallel graph reduction machine, ALICE, Darlington and Reeve [1981], was initially developed around HOPE although it now encompasses logic as well as more conventional languages. Ian Moor has written a compiler for HOPE to ALICE Compiler Target Language that is written entirely in HOPE and handles the whole language, Moor [1982].

ML, a largely functional language, was developed at Edinburgh in connection with the LCF program proving project, Gordon et al [1977] and has been implemented on a VAX by Luca Cardelli. The polymorphic type checking algorithm employed in HOPE was developed initially for ML, Milner [1977].

David Turner has developed a higher order functional language KRC (Kent Recursive Calculus) (a development of his earlier SASL (St Andrews Static Language) and pioneered a novel implementation technique for functional languages involving compilation to a simple low level order code that is itself a declarative language, Turner [1979]. Aurthur Norman and colleagues at Cambridge have built special hardware to run this order code, Clarke et al [1980] and Turner is developing a micro-coded implementation of it on a powerful mini-computer.

LISP initially started out as a pure functional language although its purity was soon destroyed by enthusiastic implementors eager to extend the language. Recently interest has been shown in reviving the pure form. Peter Henderson at Oxford has developed a pure, highly portable version of LISP called LISPKIT. LISPKIT has recently been transported to a PERQ with a section of the abstract machine interpreter micro-coded with impressive gains in efficiency.

Other centres with interest in functional languages in the U.K. include East Anglia (Burton and Sleep), Newcastle (Treleaven), Manchester (Gurd and Watson), Warwick (Wadge) and Westfield College London (Hankin and Glazier) and Queen Mary College London (Abramsky).

In the U.S. there is a lot of interest in functional languages centred around Backus's FP and FFP systems following his very influential ACM Turing Award Lecture, Backus [1978]. Many universities have active groups centred around the language and are designing parallel machines based on functional languages. It is known that several U.S. manufacturers have functional language machines under consideration at least in their research laboratories.

2.2 Logic Programming

PROLOG was first designed in France by Colmerauer, and implemented by Roussel, Roussel [1975], although a lot of the early work on logic and logic theorem proving ws done at Edinburgh. A very efficient and influential implementation of PROLOG was done on the SERC's DEC-10 at Edinburgh by David Warren, Warren [1977]. Implementation of PROLOG on a PDP-11 was also carried out at Edinburgh, Mellish [1980].

The other main U.K. centre for PROLOG is Imperial College. Here a variant of PROLOG known as IC-PROLOG was implemented in Pascal, Clark and McCabe [1979]. McCabe has also developed a version of PROLOG for 8 bit micro's known as micro-PROLOG which has been very influential in spreading PROLOG and the ideas of logic programming, he is now developing a PROLOG abstract machine with the aim of producing a highly portable PROLOG compiler. Also at Imperial Keith Clark and Steve Gregory have developed a variant of PROLOG known as PARLOG aimed at parallel evaluation and operating system style applications, Clark and Gregory [1981]. Gregory has developed a compiler for PARLOG into ALICE Compiler Target Language. Krysia Broda, also at Imperial, has written an interpreter for PROLOG in HOPE that will accept micro-PROLOG syntax and run, in parallel, on ALICE.

Other U.K. centres with an interest in PROLOG and logic programming include Exeter (Campbell), Sussex (Mellish, Sloman) and York.

A lot of interest in PROLOG has been demonstrated in Hungary where a VAX based implementation, M-PROLOG has been developed, Szeredi [1977]. In North America the University of Waterloo has an active PROLOG group that have developed Waterloo PROLOG for IBM machines. Recently PROLOG has been attracting much attention in the U.S. particularly in the areas of natural language parsing. Of course there is a great deal of interest in PROLOG in Japan which we will discuss below.

3. **Applications**

It would be fair to characterise declarative languages as in the transition stage between experimental small scale applications and large scale serious usage. Enough non-trivial applications have now been written to begin to justify the claims made for these languages on theoretical grounds but more work, both practical and theoretical, needs to be done to finally establish these languages as serious contenders to replace the more conventional languages in all applications.

HOPE has been used in Edinburgh to write sophisticated mathematical and program specification packages, Burstall [1979]. Also at Edinburgh Martin Feather used a precursor of HOPE to fully specify the text formatter from Kernighan and Plauger [1976] and transformed this, completely mechanically, to an efficient implementation, Feather [1979].

At Imperial Moor's HOPE in HOPE compiler is a large HOPE program (3,000 lines) that handles all the stages of parsing and code generation completely declaratively. The manner of its construction bore out many of our hopes for the functional languages. Our meta-language based transformation system is written entirely in HOPE as are all the tools we are developing for the ALICE programming environment such as a structure editor.

PROLOG has been extensively used at Edinburgh. The Mecho project, Bundy [1979], used PROLOG to develop a mechanics problem solving system while Pereira and Warren used it to implement a sophisticated natural language interface, Pereira et al [1976].

At Imperial PROLOG has been used very widely. Clark, Hammond and McCabe have been using PROLOG to implement expert systems. Systems have been constructed that deal with such diverse topics as supplementary benefit entitlements, dam construction, genetic engineering and care of the terminally ill, as well as the development of software tools to assist the building of such systems. Ennals and Kowalski are conducting a project using PROLOG in schools to teach not only the ideas of logic programming but the organisation of knowledge in many areas of the school curriculum, including history, language and science. PROLOG has also been used in compiler writing, Moss [1980], graphics, Julien [1982], critical path analysis, Kriwaczek [1982] and formalising legal concepts, Sergot [1982].

Much of the implementations for the PARLOG ALICE compiler and the PROLOG abstract machine are being written in PROLOG.

Hungary has some very serios applications involving PROLOG including symbolic mathematics, drug interaction analysis, architectural planning and compiler writing.

4. Program Development in Declarative Languages

Although declarative languages in themselves are much higher level and powerful than conventional programming languages their clean mathematical properties make it possible for program development and maintenance to become a much more formally based, systematic and mechanisable activity than it is at present. The idea of **program transformation** was first pioneered with the functional languages, Burstall and Darlington [1977]. The idea here is that many of the difficulties encountered at present in attempting to produce good quality software arise from trying to meet several incompatible goals simultaneously. In particular attempts to make programs clear and correct (and therefore easily modified) often conflict with the need to make them efficient. The transformation approach separates these two concerns and approaches the programming task in two stages. Firstly the programmer is encouraged to write the program initially concentrating only on making it as clear and understandable as possible ignoring any concern with efficiency. When he is satisfied with this **specification** he successively transforms it to more and more efficient versions using methods guaranteed to preserve the meaning of the original program while improving its performance. Thus we need a calculus of meaning preserving transformations. Such a calculus is almost impossible to find for conventional languages, the presence of shared structures, assignment and side effects means that the most complex checking and program provig is necessary before even the simplest manipulation can be performed. In contrast the simple substitutive properties of the declarative languages mean that such a calculus is easily produced. What is more such calculi can be proven once and for all to be correctness preserving, little or no proving being necessary when they are applied.

Transformation ideas apply equally well to the logic programming languages and have been extensively investigated, Bibel [1978] and Clark [1977]. In the functioal languages equality is used as the main deductive step in the logic languages this is replaced by logical inference.

Declarative languages have other advantages that apply at the specification stage. Although such specifications are written for maximum clarity and may employ language features that are extensions of those efficiently executed (e.g. infinite sets in the functional case and non-clausal form in the logic case) they very often can be 'run', albeit partially or slowly, to check out the initial specification. Thus a process of early prototyping is possible whereby the initial specification is itself tested and debugged and then systematically transformed to an efficient program. The crucial point is that specification and program are in the same formalism and there is a continuum, not a discontinuity, between the two. This process of software evolution, we think, matches much closer to the reality of software development than the disjoint specify - program - verify steps that the strict program verification approach prescribes.

Substantial work has already been done to show that program transformation is capable of formally but intelligably expressing large and complicated program developments and that partial mechanisation of this technique is feasible. The work reported in Darlington [1981] shows how the use of a meta-language enables the **design** of complex programs to be captured in a structured and understandable manner. This we feel has important consequences for program modification and maintenance as well as initial program development.

5. Parallel Evaluation

The benefits of parallel evaluation for declarative languages and the design of suitable machines to exploit this will be investigated more thoroughly in other companion papers. However we shall briefly discuss this topic in relation to the language's features and the ease of use of these languages.

Consider the following functional program to again compute factorial

```
factorial(n) = factb(1, n)

factb(i, j) = if i = j then 1
              else  if i + 1 = j then j
                    else factb(i, mid) * factb(mid, j)
                         where mid = |i + j|
                                     |-----|
                                     |_ 2 _|
```

Thus if were are allowed to reduce all available subexpressions in parallel an evaluation of factorial 5 would proceed

```
factorial(5) => factb(1, 5) => factb(1, 3) * factb(3, 5)

=> (factb(1, 2) * fact(2, 3)) * (factb(3, 4) * factb(4, 5))

=> (2 * 3) * (4 * 5) => 6 * 20 => 120
```

A total of 5 reduction steps instead of the 11 needed previously.

The most obvious advantage for parallel evaluation is of course the potential for dramatic increases in speed for declarative languages. Our prediction for the ALICE prototype, a relatively modest 16 processor machine, is that it will run HOPE an order of magnitude faster than conventional languages on a medium sized mainframe. Such an increase would straight away make the declarative languages very attractive indeed. However, perhaps more important is the fact that parallel machines provide a **natural** implementation for declarative languages. It has been our gratifying experience that it is easier to compile declarative languages onto a parallel machine than onto a sequential one. Furthermore often the forms of program that benefit most by parallel evaluation are the ones that are easiest to express. This is perhaps not well illustrated by the factorial example but parallel algorithms are often closer to the natural specification of problem than sequential ones. Transformation techniques can be equally well used to maximise the amount of parallel evaluation present in a program as they can to fit a program to a sequential machine.

For logic languages the question of parallel evaluation is slightly more complicated. For example consider the following classic logic program for deciding the fallibility of individuals

```
Fallible(x) if Human(x)

Human(Turing)

Human(Socrates)

Greek(Socrates)
```

Say we wanted to use this program to find a fallible Greek our query would be

    Fallible(x) and Greek(x) ?

Now there are opportunities for investigating both the Fallible question and the Greek question in parallel by using the separate clauses of the program. Thus OR-parallelism is easy to achieve. The difficulty arises in that any x that purports to be an answer to the whole query must satisfy both parts. Thus AND-parallelism gives rise to the need for co-ordination between otherwise independent parts of the program.

This question of how to achieve the maximum benefit from parallel evaluation apart the encouraging thing is that parallel evaluation allows us to implement much more fully the logic programming ideal rather than PROLOG. For instance by being inherently sequential PROLOG interpreters only provide a partial implementation of logic. That is if we view a logic program as a set of assertions PROLOG interpreters, because of their depth first backtracking nature, may go into a loop and fail to find a solution although one is logically implied by the assertions. Given a parallel machine we can implement a complete breadth first search for solutions and remove many of the impure features from PROLOG that are there just to control a sequential search.

6. Issues and Open Problems

Although much progress has been made there still remain many issues to be tackled before declarative languages can claim to be fully mature and all embracing.

6.1 Real Time Problems

The main virtue of the declarative languages, absence of side effects could be viewed as a handicap when tackling problems involving the need to effect the outside world. This is not necessarily so, but it is true that declarative languages have not been extensively used in real time or control applications. There are theoretical and language issues that need to be solved concerning the sequentialisation of operations when this is required by the application. Sequential languages are burdened all the time by explicit sequentialisation, sometimes it comes in handy but there is no reason why we cannot add it to the declarative languages when needed, the issue is how to do it cleanly and transparently.

Included in this class of problems are operating system applications. The issue here is how to specify behaviour that affects the outside world (i.e. controlling printers etc), control access to shared resources and achieve the right degree of non-deterministic behaviour. The language PARLOG, Clark and Gregory [1981] and the work of Shapiro represent substantial advances towards solving these problems.

6.2 Lack of assignment and updatable structues

The absence of assignment is the chief characteristic of the declarative languages and is essential for their power. However it is still an open question whether there are problems for which the ultimately efficient solution requires in place updating of a shared data structure. If this were so such 'destructive' assignments could be cleanly introduced into an initially totally declarative program by transformation. Programs written employing destructive assignments and side effects are notoriously error prone and hard to debug.

6.3 Parallel Evaluation

As parallel machines are only now being developed there are many interesting research problems connected with the practical behaviour of programs under parallel evaluation. What is the analog of tracing a program? How are resources allocated and controlled on a parallel machine? Is the user interface to a declarative language itself expressed in declarative terms?

7. Future Developments

The future looks highly promising for the declarative languages. The arrival of the parallel machines should see their ultimate acceptance but much can be one in the interim using conventional implementations or novel sequential architectures. The way forward seems to lie in further development of the languages, their associated programming techniques and their application to realistic problems.

One obvious development is to attempt to combine the advantages of both functional and logic languages in a new formalism. This is already hapenning to some extent, many versions of PROLOG now incorporate functional notation. Studies are also underway to incorporate unification into the functional languages. This gives a language with all the power of PROLOG with regard to invertibility etc. but still retaining the ability to use higher order functions and functional notation.

Much development work is needed before automated program development is practical but a lot of the theoretical groundwork has been done and the exposure of these techniques to large scale software engineering applications is ripe.

A great boost has recently been given to the declarative languages by the Japanese Fifth Generation Programme. This activity is now underway and reports indicate that the Japanese are enthusiastically pursuing their commitment to the declarative languages. The concentration of the Japanese activity seems to be largely on declarative languages, program specification and development systems and sequential and parallel machines for these languages. Applications at the moment seem largely restricted to using PROLOG and variants as a kernel language for system work and developing natural language interfaces.

Acknowledgements

We would like to thank all our colleagues at Imperial College and elsewhere who contributed towards this paper and the SERC and in particular the DCS programme for consistently supporting this work.

References

Backus [1978], Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Turing lecture, CACM21, 8 pp 613-641.

Bibel [1978], On Strategies for the Synthesis of Algorithms. Proc. AISB Conference on Artificial Intelligence, Hamburg, July.

Bundy [1979], MECHO: A Program to Solve Mechanics Problems. DAI Working Paper No. 50. University of Edinburgh.

Burstall and Darlington [1977], A Transformation System for Developing Recursive Programs. JACM Vol. 24 No. 1 pp 44-67.

Burstall and Goguen [1979], The Semantics of Clear, a Specification Language. Proc. 1979 Copenhagen Winter School on Abstract Software Specifications.

Burstall, McQueen and Sanella [1980], HOPE an Experimental Applicative Language. Proc. LISP Conference, Stanford, August.

Clark [1977], Verification and Synthesis of Logic Programs. Research Report, Dept. Computing, Imperial College, London.

Clark and McCabe [1979], Programmers' Guide to IC-PROLOG. CCD Rep. 79/7, Imperial College, London.

Clarke, Gladstone and Maclean [1980], SKIM, the S.K.I. Reduction Machine. Proc. LISP Conference, Stanford, August.

Clark and Gregory [1981], A Relational Language for Parallel Processing. ACM/MIT Conference on Functional Languages and Computer Architecture, Wentworth, Mass. October.

Darlington [1981], The Structured Description of Algorithm Derivations. Invited paper Amsterdam Conference on Algorithmic Languages, ed de Bakker and van Vliet, North-Holland.

Darlington and Reeve [1981], ALICE: a Multi-Processor Reduction Machine for Applicative Languages. ACM/MIT Conference on Functional Languages and Computer Architecture, Wentworth Mass. October.

Feather [1979], 'ZAP' Program Transformation System, Primer and User Manual. Rep. No. 54, Dept. of AI, University of Edinburgh.

Gordon, Milner and Wadsworth [1977], Edinburgh LCF. Report CSR-11-77, Dept of Computer Science, Edinburgh University.

Julien [1982], Graphics in Micro-PROLOG. Research Report No. DoC. 82/17, Sept. 1982.

Kernighan and Plauger [1976], Software Tools. Addison-Wesley.

Kriwaczek [1982], Some Applications of PROLOG to Decision Support Systems. Msc Thesis, Dpt. Computing, Imperial College.

Mellish [1980], The RT-11 Prolog System. Software Report 5a (revised). Dept. of AI, University of Edinburgh.

Milner [1977], A Theory of Type Polymorphism in Programming. CSR-9-77, Dept. of Computer Science, Unisersity of Edinburgh.

Moor [1982], An Applicative Compiler for a Parallel Machine. Proceedings of the Sigplan '82 Symposium on Compiler Construction, Boston.

Moss [1980], A Formal Definition of ASPLE using Predicate Logic. Imperial College, Department of Computing Research Report 80/18.

Pereira and Warren [1976], Definite Clause Grammars Compared with Augmented Transition Networks. Dept. of Artificial Intelligence, University of Edinburgh Research Paper No. 116.

Roussel [1975], PROLOG: Manuel de Reference et d'Utilisation. Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille, Luminy, September 1975.

Sergot [1982], Prospects for Representing the Law as Logic Programs. Logic Programming edited by Clark and Tarnlund.

Szeredi [1977], PROLOG Reference Manual. (Hungarian) SZAMOLOGEP, VIII. No. 3-4 pp 5-130.

Turner [1979], A New Implementation Technique for Applicative Languages. Software Practice and Experience, January.

Warren [1977], Implementing PROLOG. Research Report 39, 40. Dept. of AI Universiy of Edinburgh.

DECLARATIVE LANGUAGES

JOHN DARLINGTON

IMPERIAL COLLEGE

PROCEDURAL LANGUAGES	DECLARATIVE LANGUAGES
VON NEUMANN MACHINES	PARALLEL MACHINES
SEQUENTIAL LANGUAGES WITH ASSIGNMENT BASIC PASCAL o o o ADA	MATHEMATICALLY BASED LANGUAGES FUNCTIONAL - HOPE, KRC LOGIC - PROLOG
SOFTWARE CRISIS	FORMALLY BASED PROGRAM DEVELOPMENT
PERFORMANCE TECHNOLOGY LIMITED	PERFORMANCE ALGORITHM LIMITED
FUNDAMENTAL LIMITS	NO FUNDAMENTAL LIMITS

- DECLARATIVE LANGUAGES BASED ON MATHEMATICAL NOTATIONS

FUNCTIONAL	-	LAMBDA CALCULUS
		RECURSION EQUATIONS
LOGIC	-	FIRST ORDER PREDICATE CALCULUS

- NO ASSIGNMENT, SIDE EFFECTS OR EXPLICIT SEQUENCING

- COMPUTE BY VALUE NOT EFFECT

- POWERFUL, COMPREHENSIBLE PROGRAMMING LANGUAGES

- FORMALLY BASED PROGRAM MANIPULATION POSSIBLE

- PARALLEL EVALUATION NATURAL

FUNCTIONAL LANGUAGES

PROGRAMS SETS OF EQUATIONS DEFINING FUNCTIONS

MAX(X,Y)	=	IF X >Y THEN X ELSE Y
MAXOF3(X,Y,Z)	=	MAX(MAX(X,Y),Z)
FACTORIAL(N)	=	IF N = 0 THE 1 ELSE N * FACT(N-1)

EVALUATION REDUCTION OF AN EXPRESSION

FACTORIAL (4) $\Rrightarrow$ 4 * FACTORIAL (3)

$\Rrightarrow$ 4 * (3 * FACTORIAL (2)) $\Rrightarrow$ 4 * (3 * (2 * FACTORIAL (1)))

$\Rrightarrow$ 4 * (3 * (2 * (1 * FACTORIAL (0))))

$\Rrightarrow$ 4 * (3 * (2 * (1 * 1)))

$\Rrightarrow$ 24

- PATTERN MATCHING

  SEPERATE EQUATIONS FOR DIFFERENT CASES OF ARGUMENT

  ```
  FACTORIAL (0)     =  1
  FACTORIAL (N + 1) = (N + 1) * FACTORIAL (N)
  ```

- STRUCTURES AS TERMS IN DATA CONSTRUCTORS

  ```
  EG LISTS      NIL, CONS
        [ ]     ∿  NIL
        [1]     ∿  CONS(1,NIL)
     [1,2,3]    ∿  CONS(1,CONS(2,CONS(3,NIL)))

  (CONS(N,L)  ≡  N.L   INFIX OPERATOR)
  ```

- USED IN PATTERN MATCHING

  ```
  LENGTH (NIL)  =  0
  LENGTH (N.L)  =  1 + LENGTH(L)
  ```

- 'ABSTRACT' TYPES BECOME CONCRETE

- HIGHER ORDER - FUNCTIONS FIRST CLASS OBJECTS

```
MAP(NIL, F)    =   NIL
MAP(X.L, F)    =   F(X).MAP(L, F)
```

  - 'CANNED' ITERATORS

```
COMPOSE (F,G)  =   LAMBDA N. F(G(N))
```

  - FUNCTION LEVEL PROGRAMMING → BACKUS

- STRONG POLYMORPHIC TYPE CHECKING

- LAZY EVALUATION

LOGIC PROGRAMMING

- CALCULUS OF RELATIONS

```
EG    UNCLE(FRED, ANN)
      HEAVY(LEAD)
      TIMES(2, 2, 4)
```

- PROGRAMS SET OF CLAUSES

HORN CLAUSE SUBSET OF LOGIC

```
      FACTORIAL(0, 1)
      FACTORIAL(N + 1, V)  IF FACTORIAL(N, U)
                          AND TIMES(N + 1, U, V)

      LENGTH(NIL, 0)
      LENGTH(X.L, U) IF LENGTH(L, N) AND PLUS(N, 1, U)
```

- EVALUATION IS QUERYING PROGRAM

  (I) CHECKING

  FACTORIAL(4, 24)?

  $\Downarrow$

  YES

  (II) FINDING

  (A) FACTORIAL(4, U)?

  $\Downarrow$

  YES, U = 24

  (B) FACTORIAL(U, 24)?

  $\Downarrow$

  YES, U = 4

- INVERTIBILITY APPLIES TO STRUCTURES

  LENGTH(L, 2)?

  $\Downarrow$

  YES, L = CONS(U, CONS(V, MIL))

LOGIC PROGRAMMING UNIFIES

- DATA PROCESSING
- DATA BASES
- KNOWLEDGE REPRESENTATION
- EXPERT SYSTEM IMPLEMENTATION

ADVANTAGES OF DECLARATIVE LANGUAGES

- REFERENTIALLY TRANSPARENT NOTATIONS
  - VALUES DEPEND ON CONTEXT NOT COMPUTATIONAL HISTORY
  - POWERFUL, CONCISE, UNDERSTANDABLE PROGRAMMING LANGUAGES

- SIMPLE MANIPULATION RULES
  - PROGRAM TRANSFORMATION POSSIBLE
  - PROGRAM DEVELOPMENT FORMALLY BASED, MECHANISATION POSSIBLE

- SPECIFICATIONS 'EXECUTABLE'
  - EARLY PROTOTYPING
  - TESTING, DEVELOPING SPECIFICATIONS

PARALLEL EVALUATION NATURAL

```
    FACTORIAL(N)   =   FACTB(1,N)
    FACTB(I, J)    =   IF I = J THEN 1
                       ELSE IF I + 1 = J THEN J
                       ELSE FACTB(I, MID) * FACTB(MID, J)
                           WHERE MID = ⌊(I+J)/2⌋
```

EXHIBITS EXPONENTIAL INCREASE IN PARALLELISM

```
        FACT(5)
          ⇓
        FACTB(1, 5)
             ⇓
        FACTB(1, 3) * FACTB(3, 5)
                         ⇓
(FACTB(1, 2) * FACTB(2, 3)) * (FACTB(3, 4) * FACTB(4, 5))
                         ⇓
          (2 * 3) * (4 * 5)
             ⇓
            6 * 20
             ⇓
            120
```

U.K. IMPLEMENTATIONS

1. FUNCTIONAL LANGUAGES

    (I) HOPE

    EDINBURGH HOPE COMPILER IN POP-2

    IMPERIAL

    HOPE PASCAL INTERPRETER

    HOPE IN HOPE COMPILER

    (FOR ALICE)

    WESTFIELD

    (II) KRC

    KENT INTERPRETER AND COMBINATOR COMPILER

    CAMBRIDGE SKIM MACHINE

    (III) LISPKIT

    OXFORD

    PERQ MICRO-CODED IMPLEMENTATION

    (IV) ML

    EDINBURGH VAX COMPILER

2. LOGIC PROGRAMMING

PROLOG NOT LOGIC PROGRAMMING

(I) EDINBURGH

DEC-10 PROLOG COMPILER

PDP-11 PROLOG SYSTEM

(II) IMPERIAL

IC-PROLOG - PASCAL ON IBM

MICRO-PROLOG - 8 BIT MICROS

PROLOG ABSTRACT MACHINE

PARLOG

OR-PARALLEL PROLOG FOR ALICE

(III) YORK

PROLOG ON PERQ

(IV) SUSSEX

POPLOG

POP-2 + PROLOG

APPLICATIONS

FUNCTIONAL

HOPE IN HOPE COMPILER

TRANSFORMATION META-LANGUAGE SYSTEM

GRAPHICS

DATABASE QUERY

LOGIC

EXPERT SYSTEMS

DATABASES

TEACHING CHILDREN

LEGAL INTERPRETATION

NATURAL LANGUAGE PARSING

ISSUES

- SYNTHESIS OF FUNCTIONAL, LOGIC LANGUAGES
    - INCORPORATION OF UNIFICATION INTO FUNCTIONAL LANGUAGES
    - POLYMORPHIC TYPE CHECKING IN LOGICAL LANGUAGES
    - HIGHER ORDER FUNCTIONS IN LOGIC
- PURE LOGIC VS PROLOG

  USE OF STANDARD LOGIC
- REAL TIME/OPERATING SYSTEM PROBLEMS
- USER INTERFACE

Novel Architectures.

M.R.Sleep, University of East Anglia, NORWICH NR4 7TJ.

Table of Contents

1 Introduction

Although technology has advanced considerably since the first computers were built, the basic organisational principles have remained largely static, with the following key features:

1. Sequential, centralised control of computation via a unique sequence control register.

2. A centralised, Random Access, memory.

3. Destructive update.

These ´von Neumann´ features have served us well for over 30 years, particularly with the use of clever engineering ideas like pipelining, virtual memory, and Single Instruction Multiple Data (SIMD) extensions. These ideas, when carefully integrated and realised using the most advanced technology, have led to very powerful computers like the Cray and the DAP. An obvious first question is: why not stick with von Neumann?

1.1 Motivations for Change

The clearest motivation for re-examining the basic principles is sheer speed. Given VLSI technology, we can produce cheaply huge armies of chips to attack problems in parallel. Provided we can work out some way of organising these chips to do the work required, we can ´buy speed´ from VLSI. But the unique sequence control register in the von Neumann machine restricts us to an SIMD approach to parallelism.

A less obvious motivation is the software crisis. We want to produce high-quality software at reasonable cost. Backus [Back78] has argued that conventional languages are unnecessarily difficult to program in, and that many of the difficulties stem from a ´von Neumann´ orientation of the languages concerned. The underlying concern of a conventional programmer is to guide a single locus of control through a cunningly designed maze of assignment, conditional and repetitive statements (ie the program). At each step the programmer has (perhaps quite unconsciously) as a major concern the details of how things are done rather than getting right what is done.

Because much of our civilisation manages to stagger along using programs developed in this imperative style, it may be judged reasonably successful - at least for programming von Neumann machines with a single locus of control. Even here, however, the software crisis indicates there is something wrong with conventional languages and suggests we should examine alternatives. When 5th generation architectures with perhaps thousands of chips working in parallel are considered, the prospect of programming each chip individually becomes unthinkable, and the case for a new approach which does not require the programmer to consider individual control loci becomes overwhelming.

2 The 'Language-First' Approach to Architecture.

Although the following quote from Dijkstra [Dijk76] is taken out of context, it neatly summarises the general approach of novel architects:

"It used to be the program's purpose to instruct our computers; it became the computer's purpose to execute our programs."

The architects' starting point is now the language rather than some fiendishly clever engineering idea which takes no account of programmability. A possible disadvantage of this approach is that each language may lead to a quite individual architecture which is unsuited to other languages. In the event, just two families of languages have been considered seriously by novel architects, the lambda-based languages (eg Burge's language [Burg75], SASL [Turn79], FFP [Back78], ML [GMW79], VAL [AcDe79]) and the logic-based languages (eg Prolog [ClMe81]). Operationally, lambda-based languages require only simple (non-backtracking) pattern matching facilities and are therefore easier to support. Perhaps for this reason, and the fact that logic languages are fairly recent, the bulk of the work so far on novel architectures has focussed on lambda-based languages.

There are now signs that logic and lambda languages (and perhaps process-oriented languages too) can be integrated in a natural manner. Whilst this does not simplify the problem, it does suggest that work on lambda-oriented architectures provides a sound basis for parallel architectures which support more advanced languages.

3 The Design Process

For a given language, an idealised machine can be designed which defines operationally the semantics of the language. This Computational Model usually makes unrealistic assumptions - for example an idealised Algol machine supports arrays of unbounded size and no real computer can deal with this. The job of the computer architect is to devise a physical model which, within its limitations, behaves exactly like the computational model. The process of designing a language oriented architecture starts with the rather high level computational model and progressively refines it until it becomes physically realisable at which point it is a physical model. By the time this stage is reached, the set of programs which the model will deal with satisfactorily will be considerably smaller than the set of programs which the idealised computational model supports. Finally, the physical model is mapped onto existing technology using all the clever engineering ideas around to yield a real machine.

Given a single computational model, a huge number of differing physical models may be derived using the top-down methodology. The physical models may be distinguished both in performance terms (sheer speed) and also in terms of the restrictions placed on the programmer. A good physical model leads to real machines which run fast, and perhaps more important, do not unduly force the user to 'program round' their limitations.

No real architect uses a pure top-down methodology. In practice, there is a strong temptation to let ´efficient´ instructions on the real machine find their way into a language implementation, often changing the language semantics dramatically. Thus ´real´ LISP supports destructive assignment, and most language implementations provide ´hooks´ which allow the user to get at a relatively naked form of the raw machine.

Novel architects are not immune from this bottom-up influence, especially if they support an active user community. But the novel architect feels guilty when he succumbs to such pressures, and asks the language designer for help. A major outcome of the DCS programme has been to make the UK a world focus for this increasingly active and fruitful dialogue between language designer and architect.

4 Novel Lambda Machines.

Before describing individual novel architectures, we develop some basic ideas which (in retrospect) have guided much of the work. This task is eased because nearly all the work has focussed on lambda-based languages.

4.1 The Lambda Calculus.

The following remarkably simple syntax captures the essence of all the classical computer languages:

```
E ::=  identifier
       lambda identifier . E
       @  E  E
```

In conventional terms, functions are represented by lambda abstractions. Instead of writing:

```
f(x) = x*3
```

we write instead:

```
f = lambda x. (x*3)
```

which allows us to talk about f without worrying about naming its arguments. We ´call´ functions in the lambda calculus by applying them to an argument, eg

```
@ f 5
```

will ´send´ 5 to f, to produce the result 15 which - because it is exactly equivalent to the original expression - can replace it.

Although at first sight the lambda calculus looks rather horrid, (eg f(x)=2*x+x/3 turns into : lambda x . (@ (@ + (@ (@ * 2) x) (@ (@ / x) 3)))) the unsugared (machine) form has advantages: in particular, functions which both accept and return functions may be defined. (@ * 2) is the function which doubles its argument. In general, the 'equal civil rights' property of tha lambda calculus is a powerful mechansism for developing - in conventional terms - program forming programs, the advantages of which have been amply illustrated elsewhere [Turn81b].

Usable ('sugared') lambda languages allow the user to adopt conventional infix notation, to pre-name values of expressions using LET, and to post-name values using WHERE. Programming in a pure lambda-based language can be done in a purely descriptive fashion: we imagine the output (presumably some complex data structure) and describe it in terms of the input, using LET's and WHERE's as appropriate. Aside from the capability to write 'program forming programs' (which takes some practice), the most notable feature of programming in a lambda-based notation is the total absence of the assignment statement. This means, for example, that the usual 'loop counting' variables must be replaced by recursive calls. The reward is <u>referential transparency</u>, ie within its scope, any mention of an identifier denotes the same value <u>throughout the run of the program</u>.

<u>4.2 Computation Models for Lambda Languages.</u>

The basic rule for evaluating lambda expressions is <u>beta-conversion</u>:

```
@ (lambda x.(Ex)) F  ->   [x<-F] Ex
```

where the right hand side means (a copy of) the expression Ex with all free occurrences of x replaced by (a copy of) the expression F. This rule is simple to state, incredibly powerful, and very difficult to implement efficiently. It is also very ambiguous: in particular, given a large expression containing many reducible (ie beta convertible) sub-expressions, no evaluation order is specified.

There are two central issues in developing lambda-oriented architectures:

1. What evaluation order should we use?

2. How should beta conversion be done?

<u>4.2.1 Evaluation Order.</u>

In conventional (control flow) languages, the order in which statements are executed usually has a dramatic effect on the outcome. A major result of the lambda calculus [see eg Burg75] states (roughly) that the choice of order makes no difference to the value, although it may affect termination. Evaluation of a lambda expression proceeds by identifying one or more reducible sub-expressions (or <u>redexes</u>), and replacing them with equivalent, but simpler expressions using beta conversion. This <u>reduction</u> process is repeated until there are no more

Redexes, when the expression is in canonical form. For example, ((3*4)+(5*6)) contains 2 redexes: (3*4) and (5*6). These may be reduced in any order (or in parallel) to 12 and 30 respectively. The original expression has now been reduced to the form (12+30) which may be further reduced to the canonical form (42). Essentially, computation is viewed as controlled deduction rather than a sequence of state changes. This change of viewpoint is perhaps the most fundamental aspect of ´novel architecture´ work.

It looks at first sight as if exploiting parallelism gains speed and loses nothing. Why not ´data drive´ the computation so that all redexes are reduced in parallel? In fact, an injudicious choice of evaluation order may have unfortunate consequences:

a. it may lead to non-termination, most obviously when the two arms of a conditional statement are evaluated in parallel. Most interesting computations depend on conditional statements to prevent fruitless (and possibly infinite) computation.

b. however many chips are used, any real machine has a finite capacity for realising parallelism. Once this limit is reached, further attempts to exploit parallelism simply clog up the system queues.

c. in a distributed architecture, the communication costs involved in distributing sub-expressions to other processing elements may outweigh the time saved.

Thus the choice of evaluation order affects performance in a marked manner, and the issues noted above provide a useful checklist for evaluating novel architectures. One attractive solution is to pass the buck to the user by introducing pragmas and annotations to the language. This is reminiscent of pre virtual-memory days when every programmer worth his salt had his optimal overlay scheme for memory management. The alternative approach is to make the architecture take the decisions in a dynamic manner. This is of course the right approach, but it is much harder. At present, we cannot be sure that the distributed equivalent of ´virtual memory´ magic will appear, and certainly annotations and pragmas are useful in the short term.

4.2.2 Beta-Conversion.

The basic operational mechanism for evaluating lambda expressions is beta-conversion, which is in principle simple textual substitution. For example, using an ´outermost first´ evaluation order, the expression:

```
        f(sqrt(4)) where f(x)=  if (x=1)
                                then h(x)
                                else g(x*5)
                                fi
```

reduces (ie is beta-convertible) to:

if ((sqrt(4))=1) then h(sqrt(4)) else g((sqrt(4))*5) fi

Here we have used string reduction to realise beta-conversion, making 3 complete copies of the argument. Because the new form is a conditional, and the argument occurs in both arms, one of the copies will certainly be thrown away. Further, because of the evaluation order, two evaluations of (sqrt(4)) are involved assuming both h and g force evaluation of their arguments.

We can save much unnecessary work by copying pointers to sub-expressions rather than the full text. This graph-reduction approach, described in detail for the lambda-calculus in Wadsworth [Wads71], not only bounds the work done in copying each argument, but also allows results to be shared so that only 1 evaluation of (sqrt(4)) now takes place. But we are still making one wasted copy - and in general many more - albeit only of a pointer. In fact, we are usually doing much more copying because in a graph reduction scheme we ´peel off´ a specialised copy of the function body to hold the new pointers. Whilst ´peeled off´ copies can share common portions of the original graph, the existence of deep arguments will force much actual graph copying, even if a huge portion of the graph is later thrown away (eg in an unselected arm of a conditional).

Moving from string to graph reduction involves being progressively lazier about making copies. The standard environmental scheme for realising beta-substitution takes this process to its logical conclusion by doing no copying at all. Instead, beta-substitution is simply ´remembered´ by adding an (identifier,expression) pair to an environment. In the example above, the pair would be (x,(sqrt(4))). When the identifier x is needed for further evaluation, (eg in the conditional test (x=1)) it is looked up in the environment, and future lookups can share the benefit of forced evaluation if we take some care in the implementation.

At first sight, the environmental scheme wins hands down because copying is never done unless it is needed. In this sense, it is a purely demand-driven scheme. On closer examination, however, the picture is not so clear:

a. A fast environmental scheme must employ an efficient ´lookup´ mechanism for identifiers. In particular, deep accesses to the environment should cost little more than shallow ones.

b. by its nature, the environmental scheme shares a single copy of the expression denoted by an identifier between a potentially huge number of occurrences of that identifier. Whilst this means that all uses of the identifier experience the benefits of forced evaluation of the expression, it also means that the unique copy may act as a bottleneck in a highly parallel architecture.

c. However efficient the look-up mechanism is, it is desirable to minimise its use. In a large expression containing many occurrences of a particular identifier, it may be cheaper to perform the look-up once and distribute at least pointers, even if some of these are thrown away.

4.2.2.1 Lazy Graph Reduction and Combinators.

The newest approach to beta-substitution is to use lazy graph reduction. In Wadsworth's original scheme, every beta-substitution involved a 'full-peel' of a copy of the original graph, so that if there are N occurrences of the identifier, at least N nodes would be copied whether or not required. It would be better to do lazy copying of the graph. This may be done by examining the program at compile time and translating it into a variable-free form which replaces a tree consisting of interior nodes and leaf nodes which mention variables by a tree of interior director nodes which will switch incoming arguments to exactly the places specified by the variables. The 'switches' encode the information in the variables, which may now be replaced by anonymous holes which notionally wait to be filled by the switching process.

For example, the expression:

f(5) where f(x)=2*x+x*(3*4)

may be replaced by the tree shown in fig. 1:

fig. 1:

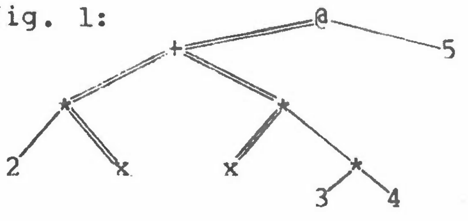
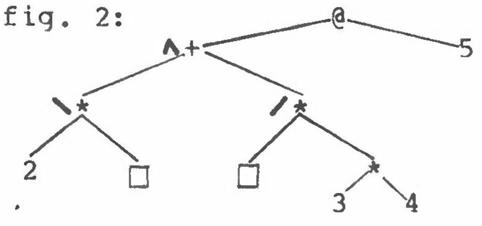


fig. 2:



The distribution sub-tree for x has been marked with double lines: it indicates that in order to evaluate the expression, the argument 5 may have to be sent both left and right at the uppermost + node in the tree, and that this + node should distribute copies both right and left. Copies (string or pointer) now arrive at the * nodes in the diagram, to be further distributed right (by the leftmost * node) and left (by the rightmost * node). Intuitively, we imagine the incoming value for x being distributed to just the places it is needed in the expression via the distribution sub-tree. An obvious encoding for distribution sub-trees is to tag each apply with a director from the set (^,\,/) representing the distribution instructions 'send both ways, send right, send left' respectively. Using this idea, fig. 1 translates to the variable-free form shown in fig. 2, where the boxes represent ´holes´ for the missing argument values. The directors guide an argument to just the places required in an expression, in a number of small steps which may be realised concurrently. Conditional expressions effectively represent directors which are determined dynamically, switching an argument left or right depending on whether the condition is true or false. The practicability of this technique was first suggested by Turner [Turn79] who introduced the S1,B1 and C1 combinators (switches) which closely correspond to the three directors. A fuller description of the director approach is available in [KeS182,Dijk80].

4.2.3 Choice of Computational Model.

It would be nice if the architect could select a preferred evaluation order and a scheme for beta-substitution in the secure knowledge that the decisions are independant. Unfortunately, this is not the case. For example, selection of outermost (lazy) evaluation favours some pointer scheme (graph reduction, lazy graph reduction or environment) as against string reduction to reduce the amount of copying. In general, string reduction is only practicable for innermost (eager) evaluators.

To complicate matters further, use of ´lazy´ evaluation [HeMo76] (which corresponds to outermost evaluation) extends significantly the class of programs which terminate: in particular, the ´lazy´ programmer can define his output using functions which operate on ´folded up´ versions of infinite lists. Because this is an extremely useful tool in the programmer´s kit, the expressive power of the language adds another dimension to the problem of choosing a computational model.

4.3 Physical Models.

An effective physical model acts as a conceptual bridge between the computational model and the hardware. At the highest level, the physical model specifies the general organisation of the architecture which can be realised in hardware.

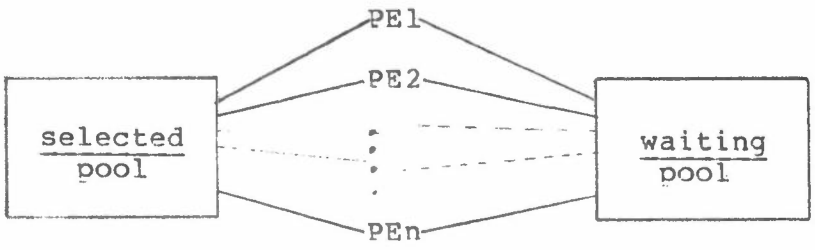
At this level, a von Neumann machine consists in essence of a processing element with one or more registers, a special sequence control register, and a global random access memory with completely destructive update. This efficiently supports sequential control flow.

To support parallel evaluation of lambda languages, a novel architecture must support the chosen computational model, which specifies an evaluation order and a scheme for beta substitution.

4.4 A General Organisation.

Any interesting evaluation order for a lambda language may create a huge number of redexes (reducible sub-expressions) which can be executed in parallel. In a von Neumann machine, every instruction appoints a unique successor, which can be recorded (indirectly) in the sequence control register. In a lambda machine, each expression may in principle appoint a large number of sub-expressions for sub-evaluation. For example, outermost evaluation of ((3*4)+(5*6)) attempts to perform the + first, and appoints the sub-expressions (3*4) and (5*6) as successors. But not only must the architecture be able to appoint more than one successor, it must also be able to remember that when all the successors have finished the parent expression may be reducible. Thus the simple ´goto´ nature of the von-Neumann architecture is inappropriate for lambda machines, which require recursive call as the basic mechanism for transferring control.

A very general organisation for achieving this is shown below:



Notionally at least, each task fully describes a sub-expression of the overall computation, together with a destination specifying where the evaluated form is to be placed. For the original expression which began the computation, this will be an output device. For sub-expressions, the destination will specify a field within some other task held in the waiting pool.

Each processing element picks any task from the selected pool and examines it to see if it requirés sub-expressions to be evaulated. If so, the relevant sub-expressions are extracted and added as tasks to the selected pool. The original task, which now has holes in it, is added to the waiting pool. These holes will be filled by returning results. If the original task does not need sub-evaluations, it is evaluated and the result used to fill a hole (either in one of the tasks in the waiting pool or in the output device). Filling in the last hole in a waiting task moves it to the selected pool.

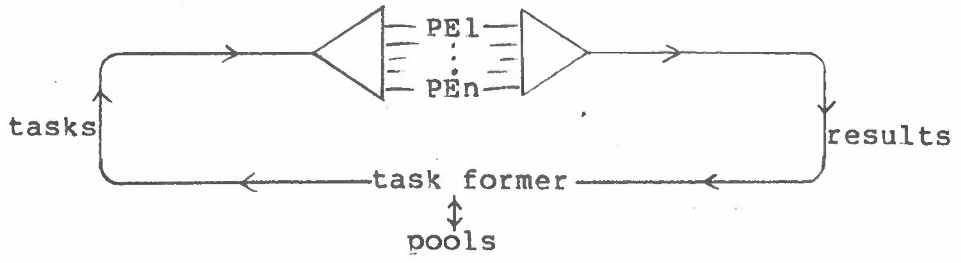
This simple picture is the basis for nearly all novel architectures to date, which however differ greatly in detail. The basic idea is to replace the single sequence control register in a von Neumann machine by a set of tasks selected for execution at each time step. The choice of computational model specifies a scheme for beta-substitution. If we can devise an efficient, extensible, highly parallel, random access implementation of the task pools required in the general organisation, there is no reason not to use pointer schemes for beta substitution. If, on the other hand this proves tricky, it may be better to risk some unnecessary copying to avoid bottlenecks in accessing the pools.

4.5 Particular Organisations for Distributing Work.

The general organisation shown above represents a rather high level model which lacks a considerable amount of important detail. It is obviously possible to simulate the model directly using a uniprocessor. It is much more difficult to invent a scheme which distributes the computation over real parallel hardware without creating communication bottlenecks. This is the distribution problem, which is the main issue at the physical level. We now examine some approaches to this problem.

4.5.1 Pipelined Ring Architectures. (PRA´s)

Rather than let tasks sit passively in a pool as the general model proposes, and making the processing elements pick them out, we might reverse the idea and make selected tasks move to the processing elements. Each processing element now processes a stream of incoming tasks and emits a stream of results. These results can be merged and the resulting stream processed by a task former which, in terms of the general model has access to the waiting pool and employs it to create a stream of new tasks. The ring is closed by feeding all the result streams to a fan-out mechanism which distributes the tasks to the processing elements as they become available. The PRA scheme is shown below in diagramatic form:



At this level of abstraction, no decision has been made about the representation of tasks, nor about their granularity. Note that the PE´s do not have direct access to the waiting pool in the PRA model, so that each executable task must include all the information (code and data) needed to perform the task. Perhaps for this reason tasks in working prototypes tend to be fine grain, eg (3+4).

In contrast to most of the other models of distribution, several prototype PRA´s are running now, notably the

Manchester Dataflow Machine [GuWa80]. By clever decomposition of the task former, the ring may be heavily pipelined, and many rings may be interconnected using an exchange switch for inter-ring communication. MIT´s Jack Dennis pioneered this approach , but continues to advocate a more static approach than the Manchester group [Denn80]. If inter-ring traffic can be kept low, multi-layered dataflow machines promise very high performances. By clever use of the ´colouring´ facilities in the Manchester machine, it is possible to support higher order functions [Kirk82].

The PRA model was originally proposed by Dennis, who has been a prime mover towards a ´top-down´ approach to architecture. Dennis has now been joined by Arvind [Arvi81], who is currently planning to build a 64-ring prototype using available chips within the next 3 years. Arvind´s proposal follows closely the Manchester work, but adds a special I-structure unit to each ring to handle large data structures. This alleviates the problem of having to physically process huge data structures each time an element is examined.

4.5.2 Packet Circulation Ring Architectures. (PCRA´s)

The most obvious bottlenecks in the PRA scheme are the input and output streams to and from the processing elements. One way of increasing throughput is to use many rings, and both Manchester and MIT are following this path. One alternative approach is to use a slotted ring for communication to distribute resources to processing elements. A slotted ring is simply a circular conveyor belt divided into slots. A sender places a message in the first empty slot he sees. A receiver looks out for a message addressed to him, and removes it to create a new empty slot. The practicality of the slotted ring concept has been amply illustrated by the Cambridge ring project [Need79].

The general idea behind PCRA´s is to place a number of PE´s in a circle and serve them with resources using one or more slotted rings. Messages, represented by 1 or more packets may denote for example a task, a global address, or data. A single slotted ring may be used for all communication, as with the real Cambridge ring, or a number of specialised rings may be used to distribute particular resources. Similarly, the ring servers may be highly specialised or more general purpose, eg processing elements with some local memory.

An advanced PCRA is being constructed at Imperial college by the ALICE (Applicative Language Idealised Computing Engine) group [Darl81]. The original ALICE proposal used two slotted rings, one for distributing tasks from the selected pool and one for distributing memory for new tasks. Both rings act as distribution agents for a global packet pool which merges the functions of the selected and ready pools in the general model. To avoid the merged packet pool becoming a bottleneck, use of a multi-ported memory with an advanced topology is proposed.

A particularly interesting feature of ALICE is that it can support traditional control flow concepts as well as reduction semantics. This is because the Compiler Target Language (CTL) [Reev82] retains some von Neumann features, supporting random access to a global packet memory and destructive update of packets. The early development of CTL allows software tools for ALICE to be developed in parallel with hardware construction.

A precursor of ALICE is the Newcastle GCF(Generalised Control Flow) architecture developed by Treleaven et al. [Farr79]. An earlier (hardware) use of the slotted ring idea is seen in the Texas Instruments Distributed Data Processor [Corn79] which effectively used a slotted ring to link several dataflow uniprocessors.

4.5.3 Physical Tree Architectures. (PTA´s)

Rather than use a slotted ring for communication, we might use a more advanced topology. Many proposals adopt a binary tree which is perhaps the simplest topology that gets everything close (O(log n)) together. Some proposals (eg AMPS [KeLP79]) use the tree structure solely for communication and load balancing purposes, with all the work being done at the leaves. Other proposals (eg DDM1 [Davi79], Mago´s machine [Mago79]) use more intelligent interior nodes.

In the AMPS proposal [KeLP79], each leaf of the tree is a processor/memory element which is capable of executing tasks sequentially or in parallel, and also capable of allocating storage for new tasks. There is no global memory, although there is a uniform global address space. The internal nodes in the physical tree perform the routing required for access to non-local memory, and external communication is via specialised leaf nodes. Although the root node of a binary tree is in principle a potential bottleneck, the load balancing takes place at the lowest possible interior node so that the root node is only employed when one half of the tree is full loaded. AMPS supports outermost evaluation of the lambda language FEL [Kell82], which includes many pragmas/annotations for user control of parallelism. FEL supports a wide range of syntactic sugar. At present, AMPS exists as a sophisticated simulation vehicle.

Because the evaluator for a lambda language is essentially recursive, the idea of building a physical tree structure of processor/memory/routing elements which recursively decomposes an expression into its primitives is attractive. An early example of this approach is the DDM1 hardware at Utah [Davi79], which evaluates simple data-driven nets. A basic difficulty with this very direct approach is that whilst work can be distributed down the tree, there appears to be no counterpart of the

AMPS mechanism for passing work from one leaf to another.

A quite different way of using a tree is the Mago machine [Mago79]. This represents a simultaneous head-on attack at all the difficult problems. The machine is unashamedly string reduction - pointers are not used. Storage management is dealt with by including it in the basic machine cycle. Expressions (written in Backus's FFP notation [Back78]) are stored in a linear array of lcells which are the leaves of a physical binary tree. Each lcell contains processing power as well as memory, and lcells are connected to immediate neighbours to facilitate data movement within the lcell array. The interior nodes of the physical binary tree, called tcells , co-operate with their neighbours in a largely asynchronous fashion to achieve distributed string reduction. A computational cycle is realised by a number of waves which sweep down from the root node of the physical tree and are reflected upwards by the leaves. The wavefront may carry control and data information. During its passage up and down the tree, the wavefront encounters tcells and lcells with which it exchanges control and data information. A number (which is variable) of sweeps is required to execute a basic Mago cycle, which can be split into the following three phases:

1.The partitioning phase. This examines the lcell array to determine the innermost (reducible) sub-expressions, and allocates tcells to each such expression. An important result of Mago's work is that each tcell will never be allocated to more than 4 sub-expressions during this phase. Microcode for the operators discovered during this phase is distributed to appropriate places.

2. The execution phase. The lcells which contain a reducible expression, and the tcells sitting above them, now operate in concert to achieve distributed reduction. If the result requires more lcells than the original expression (eg if a named operator is replaced by its FFP text), further processing is delayed until the next cycle.

3. The storage management phase. Although this is achieved in a distributed fashion, involving several sweeps, it is best thought of as a global operation which entirely rearranges the text stored in the lcells to leave room for sub-expressions which grow with reduction, and to compact those which shrink. When the whole expression represented in the array of lcells outgrows its physical bounds, some of the expression overflows into virtual memory, presently via the leftmost cell [Darn82]. During the storage management phase, all execution is suspended. Once storage management is complete, another Mago cycle begins.

The whole scheme (as Darlington once commented) is rather like a petrol engine: first the reducible expressions are determined, then 'fuel' in the form of microcode is distributed, next actual reductions (computational work) takes place, and finally (during the 'exhaust' phase) unwanted lcells are reclaimed.

The Mago machine is a unique and highly original proposal. Its major features are:

a. A global machine cycle synchronised by the physical root.

b. Inclusion of storage management in the basic cycle.

c. A direct ´string reduction´ approach.

Although not realised in hardware, the well known planar layout scheme for a binary tree makes the Mago machine attractive for direct VLSI realisation. Considerable effort has been made to develop analytic techniques for performance prediction [Mago81]. This work suggests that by clever microcoding of suitable primitives an O(n*n) time for matrix multiplication is possible.

4.5.4 Virtual Tree Architectures. (VTA´s)

Parallel evaluation of a lambda-based language requires the architecture to recursively decompose an expression into its component parts (eg arithmetic operations), evaluate some of the components, and combine the results. The whole evaluation process may be regarded as growing an ´evaluation tree´ which first expands and then collapses to yield the final result. The structure of the evaluation tree is defined by the original expression, together with the evaluation order selected in the computational model. Innermost first evaluation in its purest form completely expands the process tree until all nodes represent primitive expressions (eg (3+4)) which can be directly reduced. Outermost (lazy) evaluation reduces each node until further reductions necessitate the (lazy) evaluation of sub-expressions, and only then instructs the necessary sub-trees to grow.

If the expression at the root node of the evaluation tree determined that (say) 5 sub-expressions should be evaluated in parallel, we could in principle create 5 new physical evaluators, give one sub-expression to each, and wire the 5 new evaluators to allow them to send the results to the root node. Similarly, each new evaluator might be recursively endowed with the same powers to create and wire in new evaluators as and when they are needed.

Direct hardware implementation of this scheme is unrealistic, but it is possible to simulate it using a finite, strongly connected set of physical evaluators each of which can support many nodes in the evaluation tree. Each evaluator has primitive off-loading and memory management capability. The basic idea behind the Virtual Tree approach is to wrap a possibly huge evaluation tree around a much smaller physical network. A good VTA will initially grow the evaluation tree as fast as it can, and when all the physical evaluators are busy restrict further growth of the evaluation tree to avoid overloading the physical resources (eg system queues).

Because it entirely avoids complex compile time analysis of expressions, the simplest approach to realising a VTA is to implement some sort of diffusion mechanism, which uses only local communication between physical evaluators to make decisions regarding offloading and memory management. The basis for such a scheme is a physical evaluator which, left to its own devices, simulates depth-first priority parallel evaluation. When new nodes are created in the evaluation tree they are placed on a stack in local memory, and the uppermost node is then considered by the evaluator. Suitable modifications are made to the parent node which remains stacked and will be reconsidered when its children return results. An important feature of this simple scheme (which can be seen in [BoWW81]) is that the memory required to support it is related to the maximum depth of the evaluation tree rather than (as with very eager schemes) the total size of the evaluation tree. For a balanced evaluation tree, this is O(log N) which perhaps suggests that a means of dynamic rebalancing during evaluation is desirable.

To introduce the possibility of parallel evaluation, we connect our single physical evaluator (which simulates lots of virtual evaluators) to a small number of immediate neighbours, each of which we endow with the power to steal work from the stacks of immediate neighbours. In general, allowing neighbours to steal work from the uppermost part of the stack results in fine grain diffusion, whilst the choice of lower elements on the stack corresponds to coarse grain diffusion. Note that rather than add extra work to an already overloaded physical processing element by asking it to take responsibility of offloading, we make inactive neighbours actively seek to steal tasks. In order to make good offloading decisions, each physical evaluator needs a fairly recent picture of the workload in its vicinity. This may be maintained by forcing physical neighbours in the architecture to regularly exchange loading information.

VTA work is particularly active in the UK. The University of East Anglia has developed a simulation vehicle with full-colour graphics instrumentation, which shows clearly how a simple diffusion mechanism leads to rapid and even spread of work across the physical topology and yet governs undue exploitation of parallelism which leads otherwise to huge system queues [BuSl81]. The University of Bath [BoWW81] have been developing similar ideas although with considerably more emphasis on compile-time analysis. Bath have recently reported a working hardware configuration [MaFi82]. The idea of allowing neighbouring processors to steal work has been traced to Martin [Mart80]. Both East Anglia and Bath devoted much attention in their early work to developing physical topologies which are intuitively well suited to supporting evaluation trees. In retrospect, this effort was misplaced: firstly, every truly extensible architecture has to be realised in 3-space, and secondly fancy topologies really only help the initial ´infection´ stage of the computation. For interesting (huge) problems, this represents a decreasing fraction of the run time: it is conjectured that even a ring would support many applications.

4.5.5 Shared Distributed Memory Architectures. (SDMA´s)

In a Shared Distributed Memory Architecture, a large number (eg 4K) of processing elements access a large number of memory elements via an advanced multistage switching network. Schwartz [Schw80] develops a family of extensible computers based on this idea, and illustrates a large number of applications. The NYU (New York Ultracomputer) project [GoEA82] is examining this SDMA approach in considerable detail, for example [Bian82] discusses the wireability problem and proposes a solution for 4K processing elements.

Because previous SDMA work has largely been concerned with particular applications, its potential for lambda-based languages remains largely unexplored. There appears to be considerable potential, especially for graph-reduction schemes where efficient support of sharing is very important.

4.5.6 Novel Sequential Architectures. (NSA´s)

In the short term at least, the best way to ´buy speed´ might be to realise some novel approach to beta-substitution directly in conventional hardware. The SKIM [Clar80] project takes just this view, by implementing Turner´s combinator approach [Turn79] to beta substitution directly in hardware. The performance considerably outpaces most conventional implementations, with perhaps the sole exception of the Chalmers VAX implementation [Augu82] which directly compiles equations into VAX machine code. The SKIM hardware is not as reliable as it might be, perhaps because of its low funding: one (no doubt false) rumour suggested that SKIM was funded by passing round a hat in a local hostelry. It is to be hoped that SKIM2 will receive more appropriate support.

An early direct hardware realisation of beta-conversion is the GMD lambda machine [Berk75] which uses several hardware stacks. A parallel variant, which may fairly be considered a VTA, has recently been developed by Kluge [Klug82]. A joint ICL/Oxford project has recently produced a rather fast microcoding of the PERQ which supports Henderson´s LISPKIT LISP [Hend80]. Turner at Kent is currently engaged on a similar microcoding exercise for his KRC [Turn81a] language using an Orion. KRC is notable in supporting set abstraction which, as Turner has demonstrated [Turn81b], is a very powerful language feature.

5 A Note on Logic Machines.

In lambda-based languages, computation is realised by a relatively straightforward reduction process, which involves at worst simple matching of tree-structures to determine the appropriate re-write rule. In sharp contrast, logic languages require a much more sophisticated pattern recognition capability which generates and searches a large space of patterns. For example, the following ´fact´ tells the logic machine that (x*y=20).

times(x,y,20)

and the machine is expected as a matter of routine to deduce that the pairs (1,20),(2,10),(4,5),(5,4),(10,2),(20,1) are all consistent pair values for (x,y). Intuitively, it is much easier to realise this sort of capability in a sequential machine simply because at each step in the computation each logic variable can have at most one value at any time. In a parallel machine, there is no such constraint. This makes realising AND parallelism very difficult, although the multiple solution sets dictated by the OR construct appear a natural candiate for parallelism. A key issue here is probably what clauses constitute ´reasonable´ input for a logic machine. Warren [Warr77] has shown that ´reasonable´ logic programs can be compiled into efficient code for a DEC-10 and in principle an army of chips should be able to work fast on large logic programs. But we are much further from knowing how to ´buy speed´ for logic languages than we are for lambda-languages, and it could be argued that logic languages raise quite new issues. Pollard [Poll82] has considered the problem in some detail, and the Stockholm group [Thor82] has several active logic machine projects.

6 Conclusion

´Buying speed´ from VLSI is a hard problem. If we are content with highly specialised chips, we might follow the systolic approach [Kung79] which maps a particular algorithm onto silicon. Making a potentially huge army of more general chips work fast over a wide range of problems is much more difficult. The ´novel´ approach is to start with a language with semantics that naturally permit parallel decomposition of the evaluation, and then to develop an appropriate architecture using a top-down design methodology.

As noted in Burge´s excellent introduction to lambda-based languages [Burg75], many of the developments discussed here were forseen by Landin , particularly the use of the ambiguity with respect to evaluation order to realise parallel evaluation. Although it has perhaps taken much longer that Landin hoped, his ideas are now a major influence.

At present, we have several hardware prototypes working or near completion. The Manchester Dataflow machine is the most advanced, and a hardware demonstration of the multilayering approach to increasing performance further is underway. More recent projects such as ALICE are less advanced, but benefit from the delay by being rather less tied to the operational dataflow model. Virtual Tree Architectures are in their infancy, but recent reports from Bath and East Anglia suggest they are growing fast.

The major missing elements from an otherwise broad UK attack on the problem of ´buying speed´ for lambda languages appear to be the Physical Tree (Mago) approach, and the Shared Distributed Memory (NYU) approach. Because, as noted in [Trel81], it is still too early to pick a frontrunner, it might be sensible to encourage UK groups to follow both these (and perhaps other) lines of attack.

In the short term, working hardware demonstrations of novel architectures are only to be expected given the DCS initiative. But DCS funding does not allow ´hi-tech´ architectures. It is very important to realise that early ´lo-tech´ hardware, however novel, is unlikely to beat in stopwatch terms the continually evolving (state of the art) von Neumann rival. I conjecture that moving to a distributed implementation loses at least an order of magnitude to start with, and that lo-tech may add perhaps another two. On this basis, we may need to go to configurations involving thousands of chips before real (stopwatch) speedups are realised.

In the last decade, the ´language first´ approach to architecture has produced not only some impressive early hardware prototypes, but a wealth of practical and theoretical results which suggest that we are close to breaking the von Neumann mould.

The present position is characterised by a large number of proposals, each of which looks sensible when viewed from a particular angle. What we don´t have is a single strong idea of the kind that made virtual memory work for most applications. At present, novel architects have had to be content with demonstrating ´virtual´ as opposed to ´real´ speedups, partly because available funding could not support ´hi-tech´ realisation. I believe that we should raise our sights and determine to build, within the next decade, an extensible novel U.K. architecture which runs faster over a wide range of problems (including matrix multiplication) than the best co-existing von Neumann rival. The thesis is that breaking the von Neumann mould in stopwatch terms requires a highly entrepreneurial spirit - we must accept that in order to find the best, we must be prepared to throw away lots of the worst.

7 Acknowledgements.

Thanks are due to all the workers in the field who gladly took time off to explain what they were doing, and to SERC via the DCS panel for supporting my many discussions with the most active groups. Special thanks are due to Phil Treleaven whose recent survey paper [Trel82] provided an excellent starting point for this work. The continued active support of DCS co-ordinators, past and present, has proved invaluable.

8 References.

AcDe79 Ackerman W.B. and Dennis J.B. VAL -Preliminary Reference Manual.. MIT Lab. for Computer Science Report TR-218.

Arvi81 Arvind and Kathail V. A Multiple Processor Data Flow Machine That Supports Generalised Procedures. Proc. 8th Ann. Symp. Computer Architecture, May 1981.

Augu82 Augustsson L. Functional Compiler Status Report No.1. LPM memo 24, Chalmers University, Goteborg, Sweden.

Back78 Backus J. Can Programming be liberated from the von Neumann style? Comm. ACM 21,8.

Berk75 Berkling K. Reduction Languages for Reduction Machines. Proc. 2nd. Int. Symp. Computer Architecture, Houston, Jan. 1975.

Bian82 Bianchini R. Wireability of an Ultracomputer. Ultracomputer note 43, Courant Institute, New York University.

BoWW81 Bowyer A., Willis P. and Woodwark J. A Multiprocessor Architecture for Solving Spatial Problems.. Computer Journal, V24, No.4.

Burg75 Burge W.H. Recursive Programming Techniques. Addison-Wesley.

BuSl81 Burton F.W. and Sleep M.R. Executing Functional Programs on a Virtual Tree of Processors. Proc. ACM Conf. on Functional Programming Languages and Computer Architectures, New Hampshire, Oct. 1981.

Clar80 Clarke T.J.W., Gladstone P.J.S., Maclean C.D. and Norman A.C. SKIM - the S,K,I reduction machine. Proc. LISP-80 Conf., Stanford, Aug. 1980.

ClMe81 Clocksin W.F. and Mellish C.S. Programming in Prolog. Springer Verlag.

Corn79 Cornish M. The TI data flow architectures: The power of concurrency for avionics. Proc. 3rd. Conf. Digital Avionics Systems, Fort Worth, Nov. 1979.

Danf82 Darnforth S. DOT, A Distributed Operating System for a Tree-structured Multiprocessor. Univ. North Carolina at Chapel Hill.

Darl81 Darlington J. and Reeve M. ALICE: A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages. Proc. ACM Conf. on Functional Programming Languages and Computer Architectures, New Hampshire, Oct. 1981.

Davi79 Davis A.L. A dataflow evaluation system based on the concept of recursive locality. Proc. 1979 NCC, New York, June 1979.

Denn80 Dennis J.B., Boughton G.A. and Leung C.K.L. Building blocks for data flow prototypes. Proc. 7th Ann. Symp. Computer Architecture, La Baule, May 1980.

Dijk76 Dijkstra E.W. A Discipline of Programming. Prentice-Hall.

Dijk80 Dijkstra E.W. A Mild Variant of Combinatory Logic. EWD735.

Farr79 Farrell E.P., Ghani N. and Treleaven P.C. A Concurrent computer and ring-based implementation. Proc. 6th Ann. Symp. on Computer Architecture, New York, April 1979.

GMW79 Gordon M.J., Milner A.J. and Wadsworth C.P. Edinburgh LCF. Lecture Notes in Computer Science, Springer-Verlag.

GoEA82 Gottlieb A., Grishman R., Kruskal C.P., McAuliffe K.P., Rudolph L. and Snir M. The NYU Ultracomputer — designing a MIMD, Shared-Memory Parallel Machine. Courant Institute, New York University.

GuWa80 Gurd J. and Watson I. A Data Driven System for High Speed Parallel Computing. Computer Design, June and July 1980.

Hend80 Henderson P. Functional Programming: Application and Implementation. Prentice-Hall.

HeMo76 Henderson P. and Morris J.M. A Lazy Evaluator. 3rd. Symp. on the Principles of Programming Languages, Atlanta, Jan. 1976.

Kell82 Keller R.M. FEL - Function Equation Language - users manual. Univ. of Utah.

KeLP79 Keller R.M., Lindstrom G. and Patil S. A loosely-coupled applicative multi-processing system. Proc. 1979 AFIPS NCC.

KeSl82 Kennaway J.R. and Sleep M.R. Director Strings as Combinators. University of East Anglia.

Kirk82 Kirkham C. The Implementation of Functions on the Manchester Dataflow Computer. Univ. Manchester.

Klug82 Kluge W.E. (private comm.).

Kung79 Kung H.T. Let's Design algorithms for VLSI. CMU-CS-79-151, Carnegie-Mellon Univ.

MaFi82 Marti J. and Fitch J. The Bath Concurrent LISP machine. School of Maths., Univ. of Bath, Dec. 1982.

Mago79 Mago G.A. A Network of Microprocessors to Execute

Reduction Languages. Parts 1 and 2, Int. J. Comp. Inf. Sci., V8, Nos. 5 and 6.

Mago81 Mago G.A., Stanat D.F. and Koster A. Program Execution on a Cellular Computer: Some Matrix Algorithms. Univ. North Carolina at Chapel Hill.

Mart80 Martin A.J. A Distributed Implementation Method for Parallel Programming. Phillips research lab. memo 3-090 1, Eindhoven. (recd. July 80).

Need79 Needham R.M. System Aspects of the Cambridge Ring. Proc. 7th. Symp. on Operating System Principles.

Poll82 Pollard G.H.: Ph.D. Thesis, Dept. Comp. and Control, Imperial College.

Reev82 Reeve M. An Introduction to the ALICE Compiler Target Language. Dept. Comp. and Control, Imperial College.

Schw80 Schwartz J.T. Ultracomputers. ACM TOPLAS, V2, No. 4., 1980.

Thor82 Thorelli L. CSALAB Progress Report. TRITA-CS-8201, Royal Inst. Tech., Stockholm, Sweden.

Trel82 Treleaven P.C., Brownbridge D.R. and Hopkins R.P. Data-Driven and Demand-Driven Computer Architecture. ACM Computing Surveys, V14, No.1.

Turn79 Turner D.A. A New Implementation Technique for Applicative Languages. Software, Practice and Experience. V9, No. 1.

Turn81a Turner D.A. KRC Language Manual. Computer Lab., Univ. of Kent.

Turn81b Turner D.A. The Semantic Elegance of Applicative Languages. Proc. ACM Conf. on Functional Programming Languages and Computer Architectures, New Hampshire, Oct. 1981.

Wads71 Wadsworth C.P. Semantics and Pragmatics of the Lambda Calculus. D.Phil. thesis, Univ. of Oxford.

Warr77 Warren D.H.D. Implementing Prolog. DAI report nos. 39,40. Dept. Artificial Intelligence, Univ. of Edinburgh.

NOVEL ARCHITECTURES

M. R. Sleep

University of East Anglia, Norwich

THE VON NEUMANN COMPUTER

1. EACH INSTRUCTION APPOINTS A UNIQUE SUCCESSOR.
2. CENTRALISED, RANDOM ACCESS, MEMORY HOLDS PROGRAM AND DATA.
3. DESTRUCTIVE UPDATE.

$\boxed{\text{COMPUTATION VIEWED AS SEQUENCE OF STATE CHANGES}}$

WHY NOVEL ARCHITECTURES?

1. SPEED:

   A. 'VON NEUMANN BOTTLENECK' BETWEEN PROCESSOR AND MEMORY.

   B. NEED FORK INSTRUCTIONS FOR PARALLELISM.

2. SOFTWARE CRISIS:

   A. RECURSIVE CALL LESS ERROR-PRONE THAN UNCONSTRAINED GOTO.

   B. ZERO ASSIGNMENT (DECLARATIVE) PROGRAMMING LESS ERROR-PRONE THAN UNCONSTRAINED DESTRUCTIVE ASSIGNMENT.

$\boxed{\text{COMPUTATION AS CONTROLLED DEDUCTION}}$

THE LANGUAGE-FIRST APPROACH TO ARCHITECTURE

1. DON'T START FROM SOME FIENDISHLY CLEVER ENGINEERING IDEA, ONLY TO DISCOVER THE RESULT IS DIFFICULT TO PROGRAM.
2. DO START WITH A GOOD PROGRAMMING LANGUAGE, AND USE ITS SEMANTICS TO GUIDE A 'TOP-DOWN' DESIGN PROCESS.

'IT USED TO BE THE PROGRAM'S PURPOSE TO INSTRUCT OUR COMPUTERS; IT BECAME THE COMPUTER'S PURPOSE TO EXECUTE OUR PROGRAMS.'
(DIJKSTRA 1976)

CHOOSING A LANGUAGE

1. SIMPLICITY AND ELEGANCE.
2. GENERALITY.
3. POWERFUL ABSTRACTION CAPABILITY.
4. POTENTIAL FOR PARALLELISM.
5. WELL-UNDERSTOOD SEMANTICS.

CURRENT CHOICE: LAMBDA-BASED LANGUAGES.

FUTURE CHOICES:

A. LAMBDA-BASED LANGUAGES.
B. LOGIC-BASED LANGUAGES.
C. PROCESS-ORIENTED LANGUAGES.
D. SOME NATURAL INTEGRATION OF A..C.

IDEALISED 'TOP-DOWN' DESIGN METHODOLOGY

$$\begin{tikzcd}
\text{Language Semantics} \arrow[d] \\
\text{Computational Model} \arrow[d] \\
\text{Physical Model} \arrow[d] \\
\text{Hardware}
\end{tikzcd}$$

COMPUTATION AS CONTROLLED DEDUCTION

Rules			Computation Form
3*4->12		T=0:	((3*4)+(5*6))
5*6->30	⇒	T=1:	(12 + 30)
12+30->42		T=2:	(42)

1. Rule applications change FORM but not MEANING.
2. Intermediate states are all readable.
3. Rules may be applied concurrently.

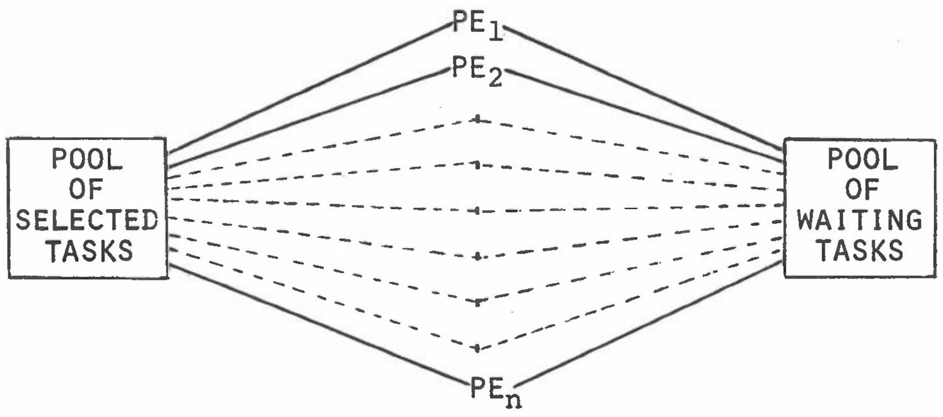
BASIC REQUIREMENTS FOR CONTROLLED DEDUCTION

1. A (PARTIAL) ORDERING RULE FOR DETERMINING THE SET OF SUB-EXPRESSIONS WHICH CAN BE EVALUATED IN PARALLEL AT EACH STEP.
2. AN EFFICIENT MEANS FOR REPLACING SUB-EXPRESSIONS WITH MORE EVALUATED FORMS.
3. (FOR LOGIC-BASED LANGUAGES): AN EFFICIENT MEANS OF DETERMINING 'UNIFYING' VALUES FOR PATTERN VARIABLES.

A GENERAL ORGANISATION

Basic Cycle

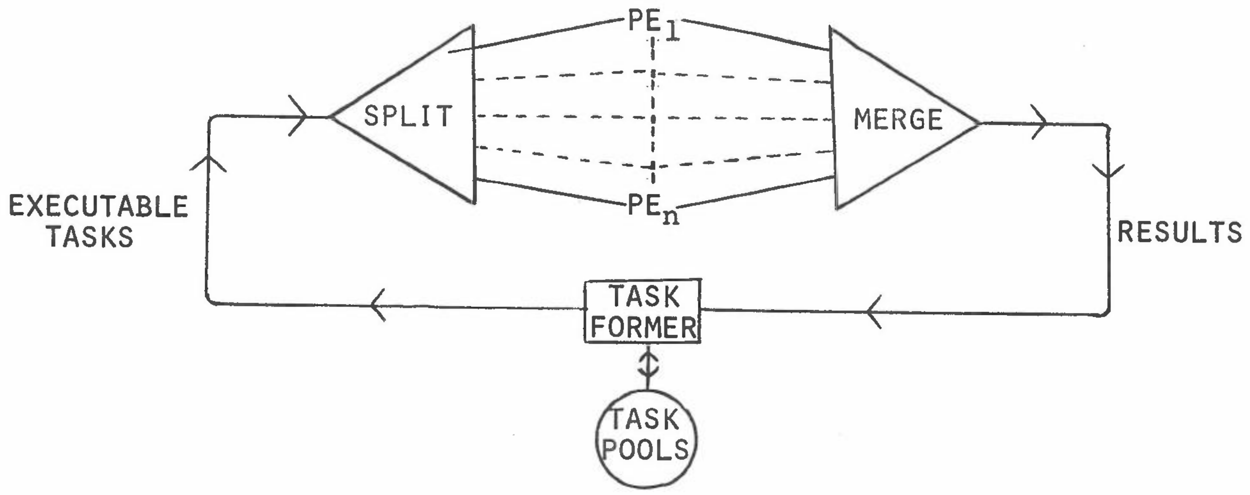
1. PE's remove tasks from the selected pool.
2. PE's process their tasks, and modify pools accordingly.



PARTICULAR ORGANISATIONS FOR DISTRIBUTING WORK

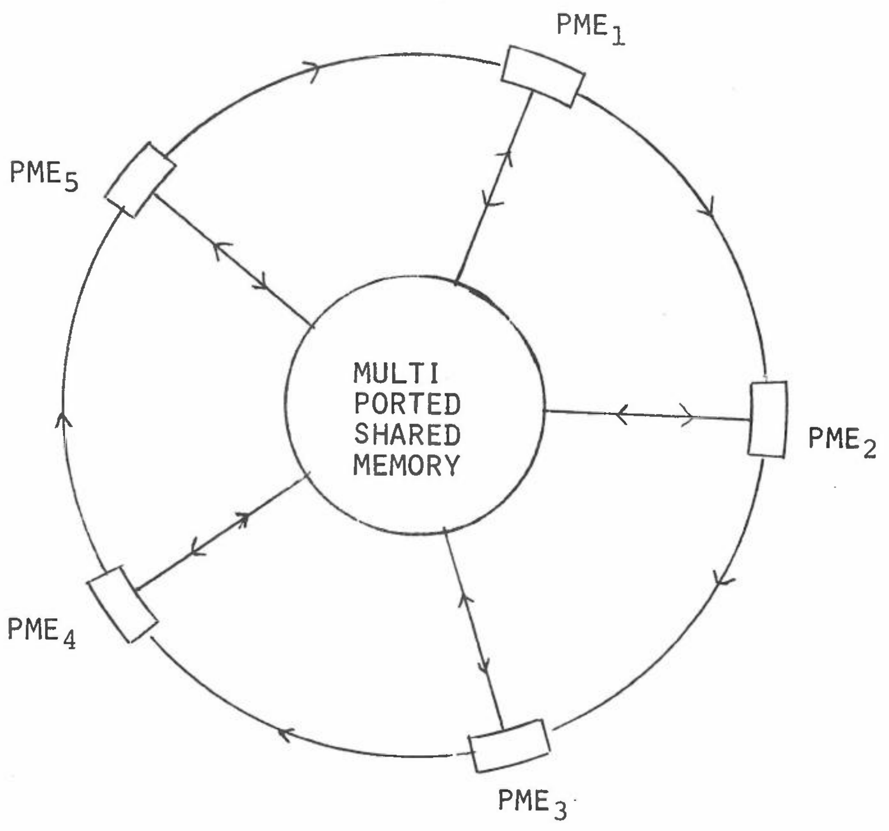
1. Pipelined Ring Architectures. (PRA).
2. Packet Circulation Ring Architectures. (PCRA).
3. Physical Tree Architectures. (PTA).
4. Virtual Tree Architectures. (VTA).
5. Novel Sequential Architectures. (NSA).
6. Shared Memory Architectures. (SMA).

PIPELINED RING ARCHITECTURES (PRA's)



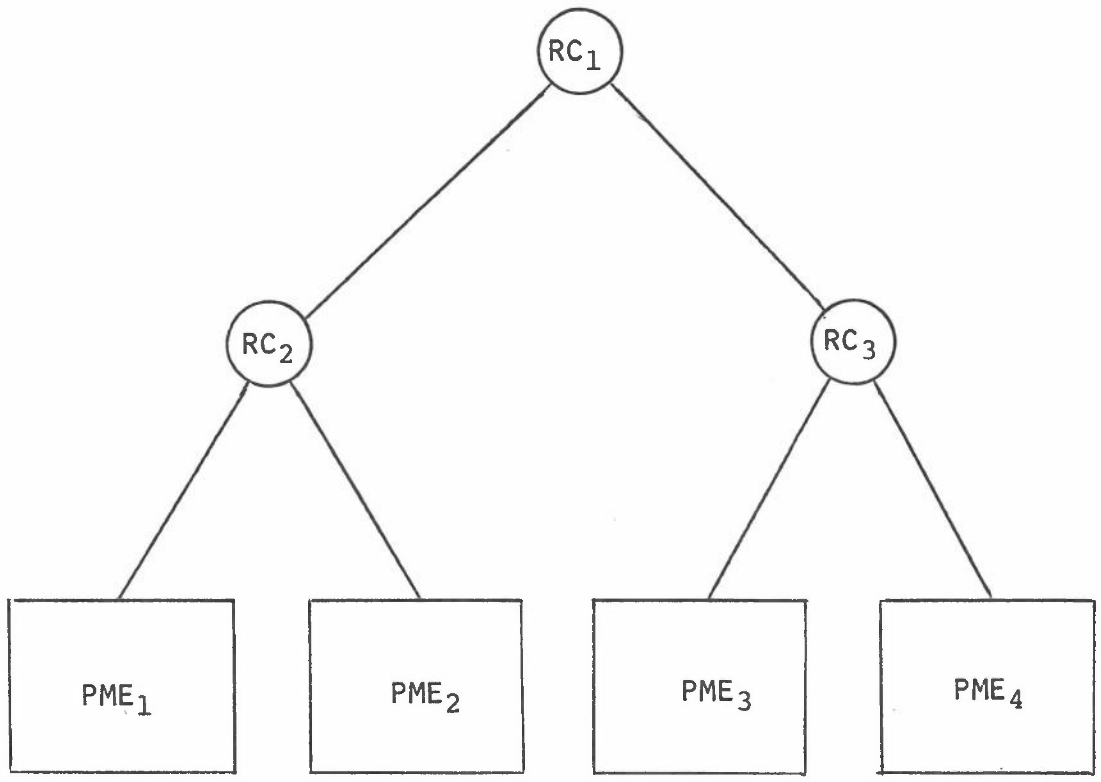
EG MANCHESTER DATAFLOW MACHINE

PACKET CIRCULATION RING ARCHITECTURES (PCRA'S)



EG IMPERIAL'S ALICE, NEWCASTLE'S GCF

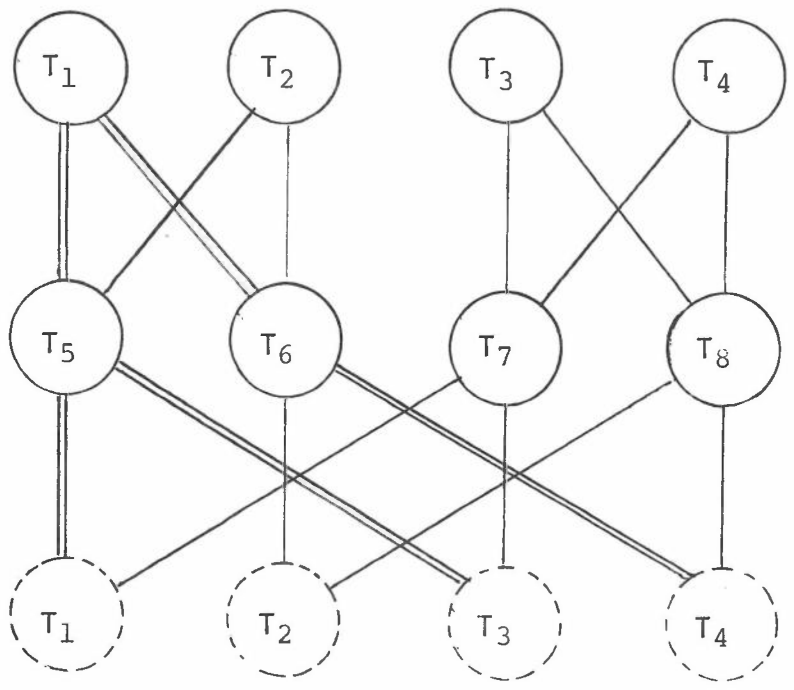
PHYSICAL TREE ARCHITECTURES (PTA'S)



EG UTAH'S AMPS, NORTH CAROLINA'S MAGO MACHINE

(RC = RESOURCE CONTROLLER)

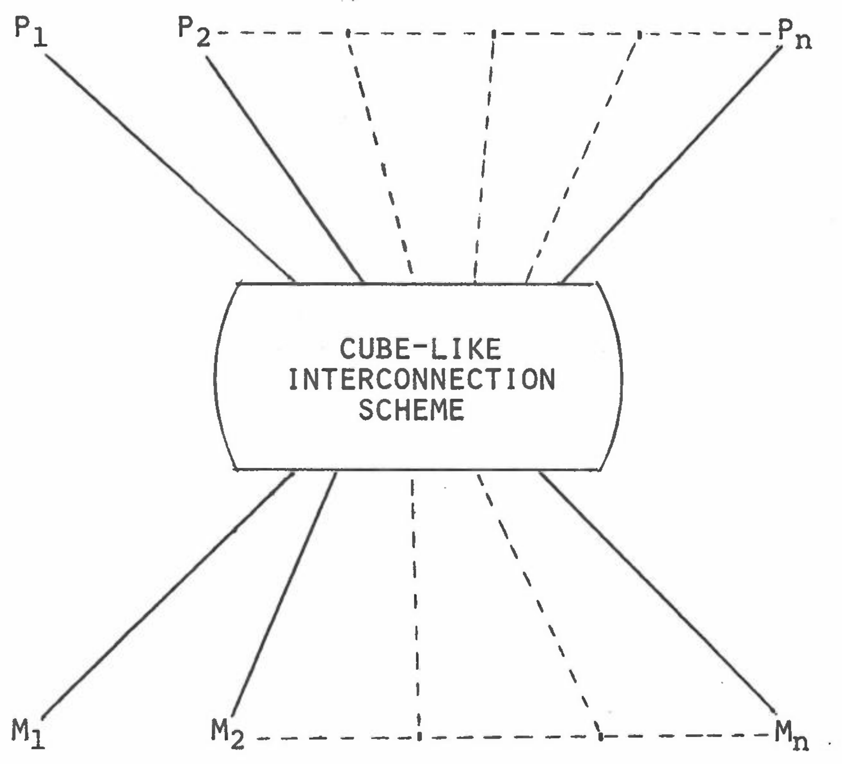
VIRTUAL TREE ARCHITECTURES (VTA'S)



EG EAST ANGLIA'S ZAPP, BATH BANK

(T = TRANSPUTER, IE PMC ON A CHIP)

SHARED MEMORY ARCHITECTURES (SMA'S)



EG NYU ULTRACOMPUTER

KEY HARDWARE PROJECTS

PROJECT	CLASSIFICATION	1ST RUN:	NO. PE'S
MANCHESTER	PRA	LATE 1981	12
ALICE	PCRA	1983(P)	16
ARVIND	PRA	1984(P)	64
MAGO	PTA	?	?
BATH	VTA	LATE 1982	6
ZAPP	VTA	1986(P)	64
NYU	SMA	?	64
TURNER	NSA	83/84(P)	1
SKIM	NSA	80/81	1

PRELIMINARY OBSERVATIONS

1. ALL GROUPS HAVE PRODUCED GOOD-LOOKING PROJECTIONS.
2. SEVERAL UK GROUPS, NOTABLY MANCHESTER, ALREADY HAVE WORKING HARDWARE.
3. CHOICE OF TECHNOLOGY (RANGING FROM CONVENTIONAL MICRO-PROCESSORS TO CUSTOMISED VLSI) MAKES COMPARISONS DIFFICULT.
4. DCS-FUNDED GROUPS ARE IN THE FOREFRONT.

EXPECTATION

A FAST LAMBDA-MACHINE AROUND THE MID EIGHTIES

KEY ISSUES

1. ARCHITECTURE:
   - A. WHICH EVALUATION ORDER?
   - B. SHOULD RESULTS BE SHARED OR COPIED?
   - C. WHICH ORGANISATION IS BEST?

2. LANGUAGE:
   - A. SHOULD WE RE-INTRODUCE CONTROL FOR SOME APPLICATIONS?
   - B. SHOULD PROCESS AND LOGIC VIEWS BE INCLUDED?

PROGNOSIS

1. Parallel hardware for lambda-languages soon.
2. Probably faster microcoded uniprocessors in the short term.
3. New languages with integrated lambda/logic/process views will appear.
4. New technology will appear, and make the Von-Neumann model even harder to beat.
5. Little chance of novel architectures winning in stopwatch terms until they are realised using state of the art technology.

DATAFLOW COMPUTER ARCHITECTURE

J.R. Gurd

Department of Computer Science
University of Manchester
Manchester M13 9PL

Introduction

It is becoming apparent that future requirements for computing speed, system reliability, software manageability and cost-effectiveness will entail the development of alternative computer architectures to replace the traditional 'von Neumann' organisation on which our present computing practices are based. Dataflow architecture is one possible alternative which aims for high-speed computing via efficient exploitation of software parallelism in a highly parallel system of processing hardware. The name 'dataflow' is derived from the graphical model of computation which is used to describe how programs are executed. In this model data is active and flows asynchronously through the two-dimensional program, activating each instruction when all the required input data has arrived. This is in direct contrast to the 'von Neumann' model in which data passively resides in store whilst instructions are executed one-at-a-time according to a defined sequence controlled by a 'program counter'.

Dataflow architectures, as described below, are only one alternative to traditional computers. Several other models with similar characteristics are emerging, and these are sometimes confused with dataflow systems, usually because they too are driven by their data. In particular, string reduction and graph reduction systems fall into this category. Such systems will not be discussed in this paper; we will concentrate on 'pure' dataflow architectures.

The paper is divided into two major sections, one covering software, the other describing hardware. In the software section we first consider the nature of software parallelism, the possible ways of representing it, and any implications for parallel machine-code design. This will provide an introduction to dataflow notation and also demonstrate the important distinction between static and dynamic systems. To conclude the software section we discuss techniques for compiling from various high-level programming languages into dataflow object-code.

In the section on hardware we consider the requirements for executing dataflow code and exploiting the exposed software parallelism. We then study three different system designs which have been, or are being, constructed as experimental research vehicles for further work applying and refining dataflow techniques.

Parallelism in Software

Two kinds of parallelism can be found in software. The first kind occurs when a common operation (or set of operations) is to be applied to many separate sets of data. An example is the element-wise addition of several arrays, as in the Fortran program:

```
      DO 10 I = 1,1,100
         F[I] = A[I] + B[I] + C[I] + D[I]
   10 CONTINUE
```

The second kind is found when different operations (or sets of operations) are to be applied to separate (or even common) sets of data. This may be found in many blocks of assignment statements, for example, the following Fortran code:

```
A = E - G
B = H * J
C = E * H + F
D = E + G
```

These forms of parallelism have been known for a long time and their importance in influencing parallel hardware design has been recognised. Flynn [12] classified hardware systems as SIMD (single-instruction-stream, multiple-data-stream) if they exploit the first kind of software parallelism, and MIMD (multiple-instruction-stream, multiple-data-stream) if they exploit the second kind.

Nowadays this classification is considered overly simple, but no generally accepted alternative taxonomy is emerging. The difficulty seems to be that parallel hardware may be deployed at a different level of 'granularity' to the obvious software parallelism. For example, in an instruction pipeline, small parts of the execution of successive instructions are processed concurrently by overlapping, regardless of any program parallelism at the instruction level, or above. In the absence of a level-independent taxonomy of parallel systems comparison of different architectures is by ad hoc methods. We have found it useful to distinguish between 'regular' and 'irregular' parallelism when comparing the abilities of dataflow systems with those of more conventional parallel systems.

Regular parallelism exists wherever the same task is to be performed many times over, usually on disjoint data. With connected data it may be necessary to exploit regular parallelism via a pipeline, as in the instruction pipeline cited above. With unconnected data, as in the case of the first (SIMD) kind of software parallelism, a lock-step parallel array of hardware can be used, as in the DAP [16] or ILLIAC IV [6]. In either case, the actions to be performed concurrently are highly regular, and the performance of the systems depends critically on whether or not the program can provide sufficient work with the required amount of the required form of regularity.

Most of the parallel computers so far constructed exploit regular parallelism of one form or another. In practice it has proved surprisingly difficult to arrange for programs to provide continuously sufficient parallelism of the desired nature. Consequently applications run at variable speed, the regular parts executing rapidly, whilst other sections are necessarily slower. In many cases the slow segments dominate overall performance and reduce the total speedup of programs to a small fraction of that intended.

Irregular parallelism exists wherever different tasks are potentially concurrently executable, sometimes on common data. This corresponds to the second (MIMD) form of software parallelism. An independent array of parallel hardware, such as in the CDC 6600 [18] (on a small scale) or the C.mmp [21] and Cm* [17] multiprocessors (on a large scale), is needed for implementation. Where common data is involved complex interlocking mechanisms are neccessary to prevent unintentional accesses being made (e.g. reading data before it has been defined, or writing before all prior reads have been completed). Note that hardware mechanisms which exploit irregular parallelism will also be able to handle regular parallelism. The reverse is not usually the case.

Few systems have been constructed to exploit irregular parallelism on a large scale, and it is in this area that many interesting experiments in computer architecture are now being conducted. The best known examples use parallelism at the 'process' level, derived from programming languages such as Concurrent Pascal [7], Modula [20], Distributed Processes [8], and Communicating Sequential Processes [14], and implemented on shared-memory or message-passing multiprocessors. Dataflow systems exploit irregular parallelism at a lower level, approximating to the conventional machine-code instruction-level.

Whether parallelism is regular or not, the key issue in developing a system to exploit it is to provide an effective notation for expressing potential parallelism in programs. In the following section we develop a notation for instruction-level irregular parallelism by examining the nature of inherent parallelism in a small segment of conventional Fortran code.

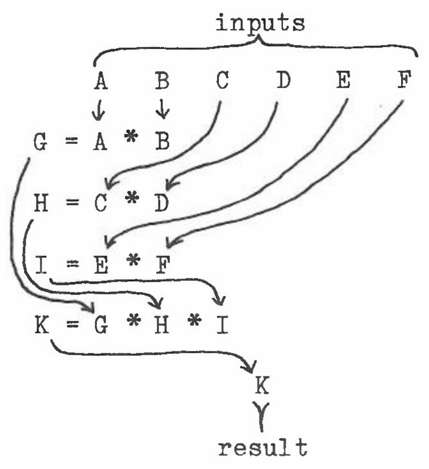
Programs as Graphs

Consider the following set of Fortran assignments which multiply together the 'variables' A, B, C, D, E and F and put the result in 'variable' K:

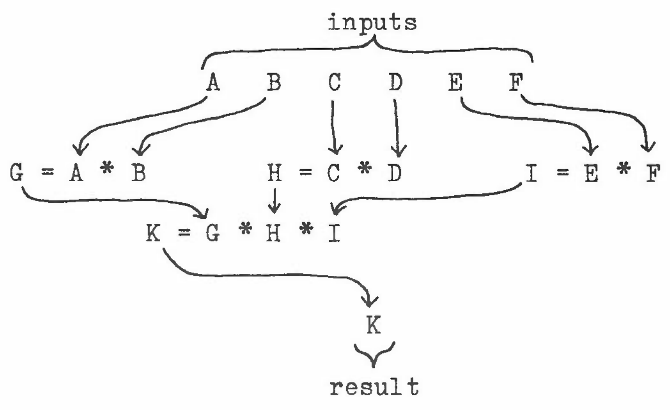
```
G = A * B
H = C * D
I = E * F
K = G * H * I
```

To discover the potential software parallelism we must discard the traditional view of a program as a list of instructions which manipulate data in fixed storage locations in a defined sequence. Instead we need to concentrate on the role the individual storage locations play as they temporarily hold data values whilst the latter pass between operations in the program. The pattern of store accesses brought about by the sequence of activation of instructions is normally contrived by the programmer to achieve the combinations of data with operators dictated by the particular problem being solved. The fact that this is specified as a one-at-a-time process owes more to the history of development of computers than to inherent constraints in the problems that computers are used to solve.

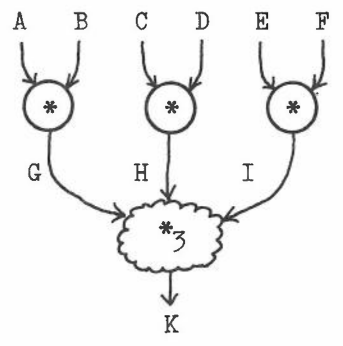
An alternative view of the combination of data with operators is obtained by constructing a data dependence graph for the program. Algorithms for this task are in common use for conventional machines in optimising compilers. In the example above, we simply draw a number of arcs over the program, one arc for each variable. The tail of an arc shows where the variable is assigned, and the head shows where the variable is consumed (by appearing on the right-hand side of an assignment statement). In more complex examples more than one arc may be required for a variable when it appears on the right-hand side of more than one assignment statement. Multiple assignments, where a variable is assigned a value at more than one point in the program, can be dealt with by systematically renaming the variables so that a version is created without multiple assignments, but with the same meaning as the original. Where variables appear only on the right-hand side they are assumed to be input data to the program segment. The resultant graph for our example is shown below:



This diagram is more visually attractive if it is rearranged to show enforced sequence down the page, with potential concurrency across the page, as follows:

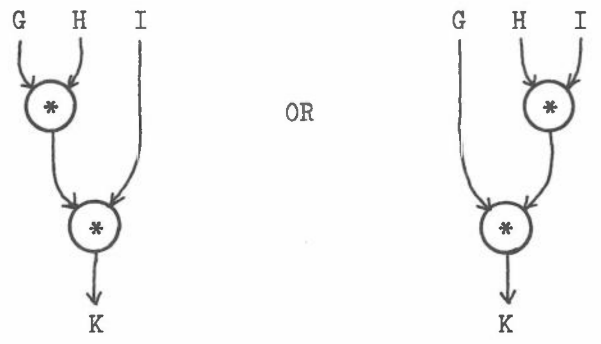


In this graphical form it is possible to omit all the variable names as they are now superfluous, being constrained to be the same at head and tail of each arc. If names are required (as an aid to understanding, or for writing a textual version of the graph), they can be written just once, alongside the appropriate arc. Each assignment statement can be simplified to a description of the expression to be computed. In many cases this will be a simple arithmetic operation, e.g. the multiplication in our example:



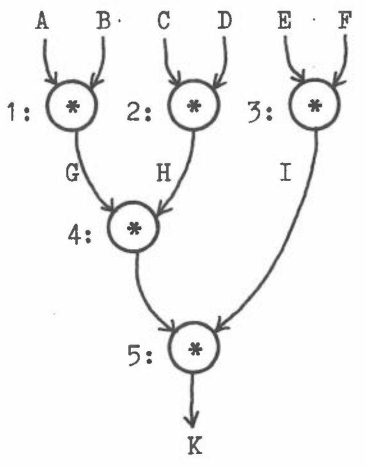
We have now constructed a simple statement-level data dependence graph. Note that it retains the meaning of the original program, but it also shows potential parallelism and enforced sequence in a two-dimensional format. In order to illustrate all the program parallelism available for exploitation by instruction-level parallel hardware it is necessary to decompose the program even further. Of course the level to which we descend is completely arbitrary. We could build a system capable of muliplying three values together in one instruction (in which case the above graph would not need further reduction), or we could go to the extreme of implementing only boolean operators (AND, OR, NOT, etc.) in hardware, and building up more complex operators using standard techniques (in which case our example graph would require considerable further decomposition). Most of the dataflow computers currently under construction use an instruction-level comparable to that of a 16-bit minicomputer with extended arithmetic capabilities. We shall assume this level in the remainder of this paper. This implies the availability of straightforward monadic and dyadic arithmetic operators on integer and floating-point numbers, and we will also assume the existence of operators which generate and manipulate boolean values.

In our example program it will be noted that the lowest expression evaluation in the graph is not a machine instruction at this level. Consequently it must be implemented by a sub-graph of instructions such as either of the following:



In this particular example it is immaterial which of these alternatives is used, and a compiler could choose between them arbitrarily. In other cases there will be good and bad options and compilers will need to be sensitive to the assessment criteria if they are to produce optimal code under a wide range of conditions. To develop such assessment criteria we need to know how programs will actually execute on a specific parallel hardware configuration. This is too difficult to discuss in detail here, but we shall finish this section with a brief description of an abstract dataflow implementation model from which the basic principles of execution may be derived.

Consider a complete machine-level program graph for our example in which each multiply instruction is given an identification number:



Remember that the purpose of this notation is to allow all potentially concurrent instructions to execute simultaneously. In the original sequential program we would expect the multiplications to be performed in the order {1}, {2}, {3}, {4}, {5}, producing the answer in five multiplication times. On the graph above we can see that either of the parallel execution orderings {1, 2, 3}, {4}, {5} or {1, 2}, {3, 4}, {5} will produce the answer in three multiplication times (given at least three and two multipliers, respectively). The problem for the parallel execution model is to cause one of these parallel execution orderings to be followed.

It is difficult to arrange activation of instructions by some parallel equivalent of a program counter. In the first place such program counters would have to be associated with processors, and the variable amounts of parallelism that could occur might require large numbers of these processors, many of which could frequently become idle. Secondly, the idea of a program counter is closely linked to the concept of a linear data store with fixed locations for each program variable. Large numbers of active instructions would imply large numbers of active store locations with attendant problems of multiplexing the required accesses. In addition to this each horizontal 'band' of instructions would have to be synchronised so that the next lower band could not start processing until all current instructions had terminated. This implies that a program would proceed at the speed of the slowest operation in each band. Apart from these problems, the task of allocating instructions to processors would be extremely difficult.

These arguments constitute a compelling reason for abandoning program counters in instruction-level parallel computers. The key to making this transition is to notice that a data dependence graph shows how <u>instructions are dependent on data</u>. It is not sensible to execute an instruction before all the data it requires is available. Conversely, once an instruction has finished executing, all other instructions that are waiting for its output data can be safely activated. The simplest way to execute a graph program so as to obey these rules is to send data directly from instruction to instruction according to the data dependence arcs, and to allow each instruction to execute when and only when it has all its required input data available. In this way the graph program execution will be <u>data-driven</u>.

We can illustrate data-driven execution of graph programs by introducing data-carriers (known as 'tokens', after Petri-net notation) onto the data dependence graph. Each token carries one data value. A token is constrained to move (at any speed it can) from the tail to the head of one data dependence arc. Tokens wait at the heads of their dependence arcs until all other arcs (if there are others) pointing to the same instruction also have tokens at their heads. At this time this instruction can be executed, taking an arbitrary amount of time to complete, after which its result token(s) is(are) placed on its output arc(s). The tokens causing the execution are no longer needed, and so they will be removed from their (input) arcs.

The sequence of 'snapshots' in Fig.1 shows how our example program could be used to evaluate 6! by sending tokens with integer values 1 to 6 to the program inputs A to F, respectively. Tokens are shown on the dependence arcs as black discs with the associated values written alongside. The way in which the data appears to flow through the program graph during execution is the reason for the name 'dataflow'.

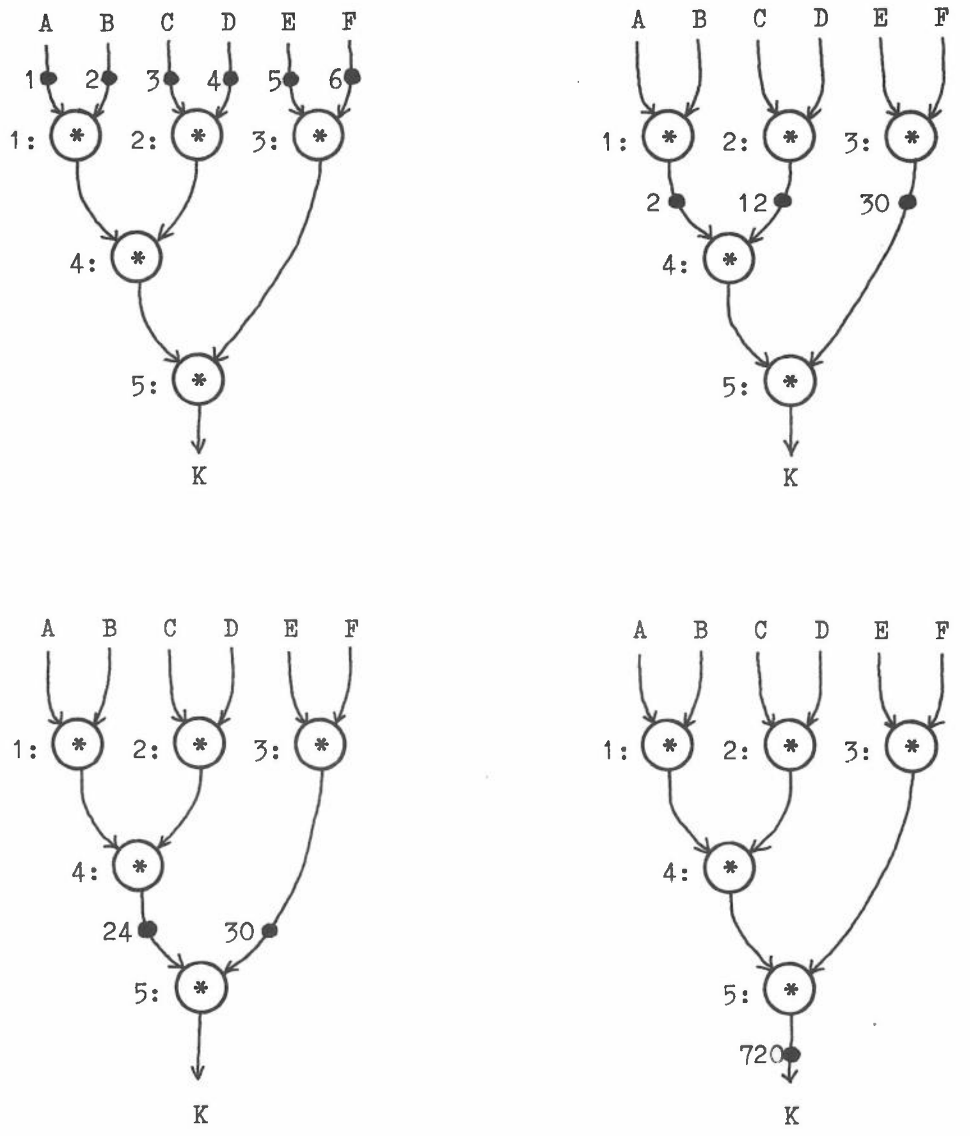


Figure 1. Dataflow evaluation of 6!

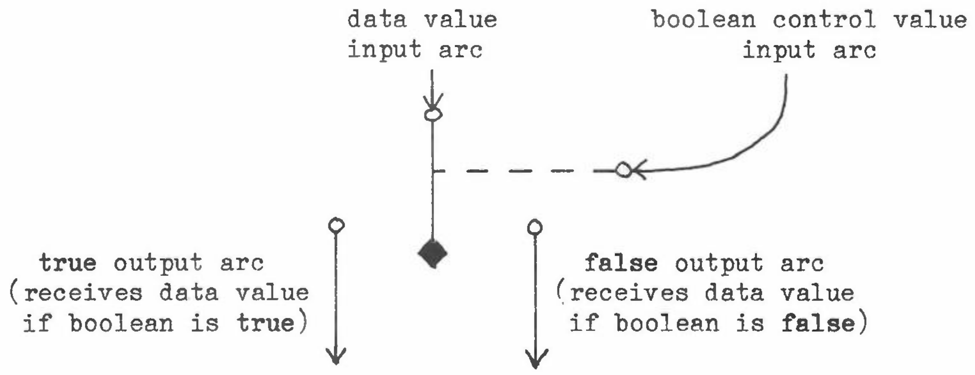
Generalised Dataflow Graphs

The multiplication program considered above is not a general example of conventional computing practice. The only arithmetic operation used is multiplication and there are no control structures, such as conditionals or loops. In this section we consider enhancements to the dataflow notation introduced above which help to accomodate more general programs.

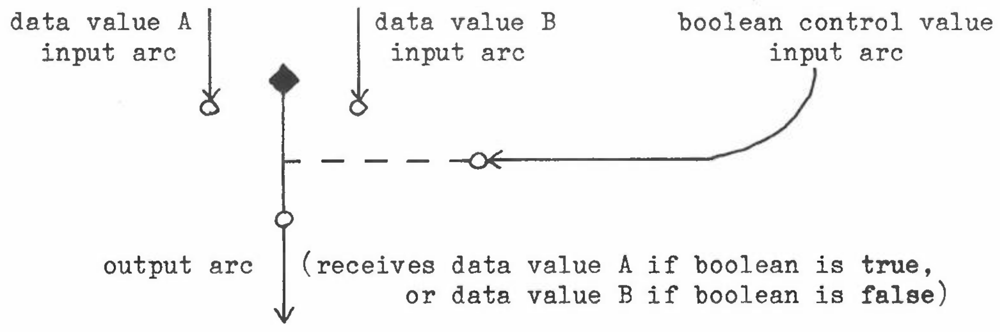
The first point to be made is that any form of machine instruction can be represented by a node in a dataflow graph, and, therefore, could be executed in parallel with other instructions. This property makes the graph notation useful for exploiting irregular software parallelism. The simplest case in which this is advantageous is in the evaluation of general arithmetic expressions in which any arithmetic machine instructions could be used. Such expressions can be easily converted into graphs. In fact most conventional compilers already generate 'expression evaluation trees', when parsing high level programs, before they generate the required linear object code. The dataflow execution model demonstrates how such trees may be evaluated directly, in time proportional to their height, using parallel instruction execution. At a higher level, the model also allows whole expressions to be evaluated concurrently. Additional parallelism can be found when control structures are invoked.

The simplest control structure is the conditional (**if** ... **then** ... **else** ... **fi**). We can construct a data dependence graph for a conditional statement using conditional dependence arcs which are controlled by the runtime evaluation of a boolean predicate. These arcs are implemented using two 'switching' machine instructions known as branch and merge. These may be visualised as two-way switches inserted into the arcs of a standard dependence graph. Each switch selects one of two possible routes for an incoming data token, the other route being left inactive. The route is selected according to the value of a boolean control token. The data and control tokens wait for each other at the inputs to the switch exactly as they would at a dyadic or triadic arithmetic instruction:

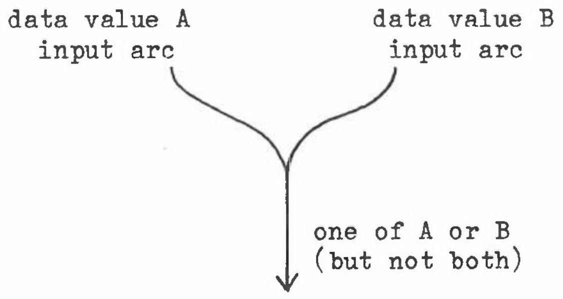
Branch


data value input arc
boolean control value input arc
true output arc (receives data value if boolean is **true**)
false output arc (receives data value if boolean is **false**)

Merge


data value A input arc
data value B input arc
boolean control value input arc
output arc
(receives data value A if boolean is **true**, or data value B if boolean is **false**)

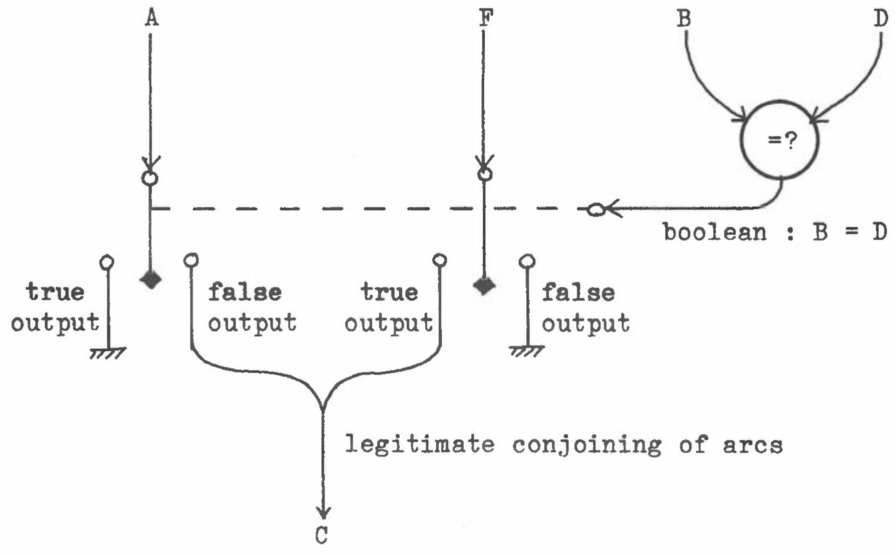
Where it is certain that only one of the data inputs to a merge instruction will be generated, and in proper correspondence to the associated boolean (e.g. from a previous branch instruction using the same control value), the merge may be omitted from the machine code and the two data arcs conjoined:



Using the extended instruction set we can implement a conditional Fortran statement such as:

```
C = A
IF (B.EQ.D) C = F
```

as the following graph:



where ⏚ indicates that tokens travelling down this arc will be destroyed, and the '=?' instruction generates a boolean value showing whether its two data inputs are equal.

Switch instructions are most powerful when used to implement graphical loops and functions. These are important because they allow complex computations to be defined by relatively small programs, in the same way as conventional loops, subroutines or procedures. However, these reentrant constructs pose substantial implementation problems in a parallel computer because of the possibility of simultaneous activation of the reentrant code.

Dataflow researchers have discovered three methods for dealing with these implementation problems. The first method is the simplest, but it is not completely general because it prohibits concurrent reentrancy. This solution allows only graphical loops which are reactivated in strict sequence. Although a limited amount of parallelism can be obtained by pipelining within the cycles of a loop, there is often further parallelism which can only be extracted by a more general scheme. Systems which implement this first scheme, allowing only sequential, cyclic reentrancy, are known as _static_ dataflow systems.

The alternatives are known as _dynamic_ dataflow systems. For example, the second scheme permits concurrent reentrancy via an _apply_ instruction which, every time it is activated, creates a new copy of the reentrant part of the graph it controls. All input tokens to this copy are gathered together at the _apply_ instruction and are then transferred to the unique new copy of the reentrant code. An _exit_ instruction, placed at the end of the copy of the reentrant code, gathers together all the output tokens for the activation and transfers them back to the output arcs of the appropriate _apply_ instruction. The copy of the reentrant code is then destroyed. This scheme is called the _dynamic code-copying_ scheme. Its operation is analogous to conventional macro-expansion in that extra code and data space is allocated whenever it is called for, in order to avoid data sharing code concurrently.

On the other hand, the third scheme allows data to share code by 'tagging' tokens as they enter into and exit from the reentrant areas. This system is similar to the use of a stack for implementing procedures and functions on conventional machines, except that the concurrent activation of shared graph code requires that each token be individually tagged with the appropriate 'name-base' instead of using a global stack register to identify the currently active data space. In visual terms tagging can be thought of as the process of colouring the data tokens. The graph execution rules need to be modified so that only tokens of the same colour (i.e. those carrying identical tags) can group together to cause execution of an instruction. Special instructions are needed to create new tags at entry to, and to restore old tags at exit from, the reentrant code. Tokens must carry extra bits to denote the tag. This scheme is known as the dynamic tagged, or dynamic code-sharing scheme.

Hybrid dynamic systems use both code-sharing and code-copying, to reduce the size of tag required.

Structured Data

Compact programs are also achieved using data structures by which a single variable name refers to a large collection of simple data items. Two schemes have been developed to implement data structures in dataflow graph programs.

The first scheme uses separate storage to hold the structures and represents each structure travelling in the program graph by a pointer token. The specialised structure store is responsible for executing read and write operations on structures, and also for issuing the appropriate pointers. All other instructions are as described above, and operate on scalar data, or control the flow of pointer tokens through the program graph.

An alternative scheme uses the tagging system described in the previous section. Each element of a data structure is a normal token which carries a unique tag defining the position of the element in the structure. Tag-sensitive instructions are used to manipulate the structure in the required way. This scheme is particularly useful for implementing regular structures, such as arrays, whose elements are all subject to continuous processing (as, for example, in signal processing applications).

Compilation of Graph Code

The examples introduced earlier demonstrate that it is possible to generate dataflow graphs from a conventional high-level programming language such as Fortran. However, the analysis algorithm that forms data dependence graphs from these languages is highly complex and takes a long time to execute. There exist other languages which are much easier to translate and these are receiving the majority of attention in dataflow research projects.

In this context, the single-assignment languages (SALs) are important. They have no concept of sequential execution and no direct control statements such as the GOTO. To combat the ambiguities that might arise from reassigning values to variables, the languages allow each variable to be assigned just once in a program. Constructs which permit controlled reassignment in special cases, such as loops, are provided. SALs tend to use the data structures, such as arrays and streams, that can be readily implemented in dataflow graphs. There are often strict type and scope rules. In particular, it is common to prohibit all forms of side-effect in reentrant constructs. The net results are languages that provide ideal textual syntax for the description of dataflow graphs. However, many SALs were developed without reference to dataflow execution, and they are also very similar to the _functional_ or _applicative_ languages which have been developed without reference to any means of execution at all.

Functional languages are based on the mathematics of functional algebra and have no concepts of storage state and assignment. They are sometimes referred to as _zero-assignment_ languages. In fact, if assignment is restricted to occur only once for each variable in a program, the effect is the same as if there were no assignment at all and 'assignment' statements were treated as _definitions_ of the variables. In this sense SALs and functional languages are identical and it should come as no surprise to find that absence of GOTOs and side-effects are common to them both. However, functional algebra allows more powerful programming constructs than most SALs because they permit construction of higher order functions and comprehensive data structures. Consequently the two groups are not directly equivalent. Nevertheless they have enough in common to make it highly probable that functional languages will be amenable to efficient implementation on dataflow systems. Research is in progress to demonstrate operational compilers in this area.

Summary of Dataflow Graphs

Dataflow graphs are a convenient notation for representing parallel computations. They permit conditional constructs, loops, functions (including recursion), and data structuring. Translation to dataflow graphs is feasible from a wide range of high-level programming languages.

There is a natural classification for dataflow systems according to the way they handle reentrant code. The three classes of system are known as _static_, _dynamic code-copying_, and _dynamic tagged_ schemes.

Parallelism in Hardware

Before considering possible hardware implementations for a dataflow computer, it is worth identifying two fundamental parallel hardware configurations and summarising their characteristics. The configurations are known as the pipeline and the parallel array.

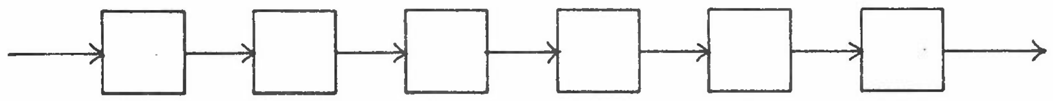
Pipelines are used where each hardware task (for example, an instruction) can be subdivided into several shorter tasks that can be executed in sequence and, preferrably, in isolation. Parallelism is exploited by overlapping the operation of successive pipeline stages on successive instructions. This situation is illustrated for a six stage pipeline in Fig.2. The stages of the pipeline are shown across the top of the diagram and the optimum distribution of instructions through the hardware over a period of time is shown underneath. After an initial delay of 6t results are produced at the output every period t, even though each complete task takes 6t to complete. To achieve this speedup the following assumptions have been made:

(i) each subtask takes an identical, relatively short time to perform;

(ii) there are no data dependences between the modules apart from the main path through;

and (iii) there are no external constraints on providing input to the pipeline, or to disposing of the results at the required rate.

If any of these assumptions is invalid, the distribution of tasks through the pipeline will be uneven and the average rate of processing will decrease.



stage	1	2	3	4	5	6
time						
0- t	task 1	-	-	-	-	-
t- 2t	task 2	task 1	-	-	-	-
2t- 3t	task 3	task 2	task 1	-	-	-
3t- 4t	task 4	task 3	task 2	task 1	-	-
4t- 5t	task 5	task 4	task 3	task 2	task 1	-
5t- 6t	task 6	task 5	task 4	task 3	task 2	task 1
6t- 7t	task 7	task 6	task 5	task 4	task 3	task 2
7t- 8t	task 8	task 7	task 6	task 5	task 4	task 3
8t- 9t	task 9	task 8	task 7	task 6	task 5	task 4
9t-10t	task 10	task 9	task 8	task 7	task 6	task 5
10t-11t	task 11	task 10	task 9	task 8	task 7	task 6
11t-12t	task 12	task 11	task 10	task 9	task 8	task 7
12t-13t	task 13	task 12	task 11	task 10	task 9	task 8
etc.	¦	¦	¦	¦	¦	¦

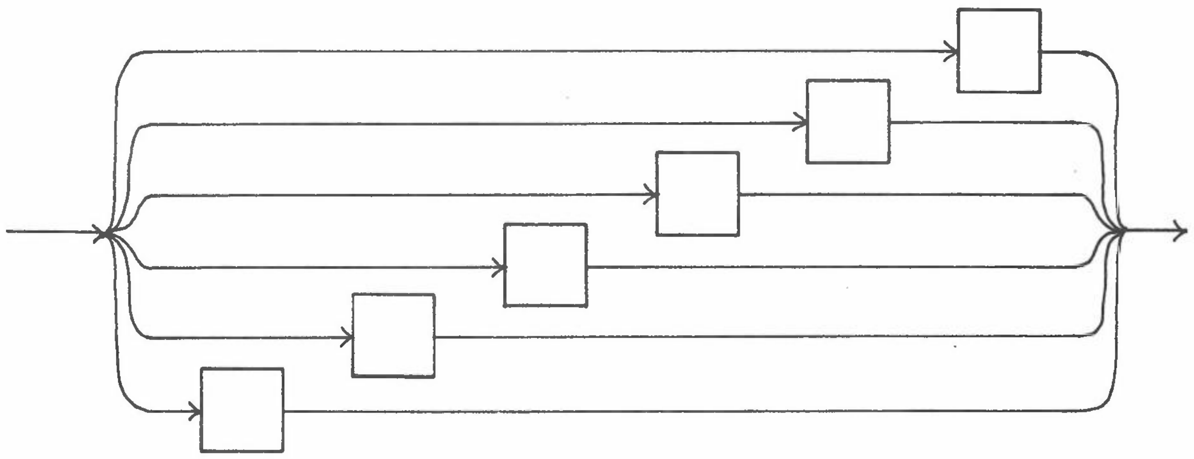
Figure 2. Six stage pipeline.
(delay per stage = t)

Parallel arrays are used where each hardware task cannot easily be subdivided, but there are many tasks independently available for execution so that they may be distributed and executed concurrently. A parallel array may be activated serially or in parallel.

In a serially-activated system the parallel array is located in a pipeline and is fed with tasks at a relatively high rate, each task being sent to a different module until all modules are active. The number of modules required is roughly the time per task divided by the time per input from the pipeline, so that the first module to be filled with a task is emptied to the output pipe just as the last module is receiving its input. This situation is illustrated for a six module array in Fig.3. Note that finished tasks are output from the array at the same rate as for the pipeline above, and after a similar initial delay. The following assumptions have been made:

(i) each task takes an identical, relatively long time to perform;
(ii) there are no data dependences between the tasks or modules apart from the input/output ports;
and (iii) input and output occur at an appropriately fast rate.

If any of these assumptions is invalid the performance again deteriorates.

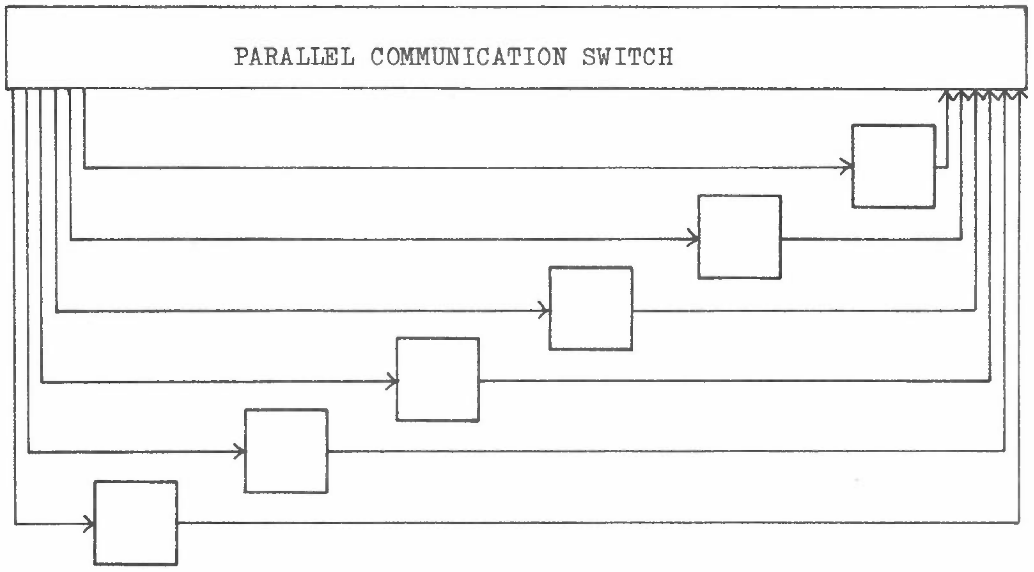


module	1	2	3	4	5	6
time						
0- t	task 1	-	-	-	-	-
t- 2t	task 1	task 2	-	-	-	-
2t- 3t	task 1	task 2	task 3	-	-	-
3t- 4t	task 1	task 2	task 3	task 4	-	-
4t- 5t	task 1	task 2	task 3	task 4	task 5	-
5t- 6t	task 1	task 2	task 3	task 4	task 5	task 6
6t- 7t	task 7	task 2	task 3	task 4	task 5	task 6
7t- 8t	task 7	task 8	task 3	task 4	task 5	task 6
8t- 9t	task 7	task 8	task 9	task 4	task 5	task 6
9t-10t	task 7	task 8	task 9	task 10	task 5	task 6
10t-11t	task 7	task 8	task 9	task 10	task 11	task 6
11t-12t	task 7	task 8	task 9	task 10	task 11	task 12
12t-13t	task 13	task 8	task 9	task 10	task 11	task 12
etc.	|	|	|	|	|	|

Figure 3. Six module serially-activated parallel array. (delay per module = 6t)

In a parallel-activated array each module executes tasks independently, and output from each module enters a routing network which sends data to the input of the next required module. Perhaps the easiest way of visualising this is to imagine all the modules locked in step with one another, executing their different instructions literally in parallel. The distribution of tasks in a six module parallel-activated array is illustrated in Fig.4. The six tasks start and finish each period of 6t time units, giving the same average throughput as the structures studied above (N.B. similar assumptions have to be made).

In the following sections we study configurations of these basic structures which implement dataflow schemes in hardware. Note that in practice it is difficult to ensure the validity of the conditions required for maximum throughput.



module	1	2	3	4	5	6
time						
0- t	task 1	task 2	task 3	task 4	task 5	task 6
t- 2t	task 1	task 2	task 3	task 4	task 5	task 6
2t- 3t	task 1	task 2	task 3	task 4	task 5	task 6
3t- 4t	task 1	task 2	task 3	task 4	task 5	task 6
4t- 5t	task 1	task 2	task 3	task 4	task 5	task 6
5t- 6t	task 1	task 2	task 3	task 4	task 5	task 6
6t- 7t	task 7	task 8	task 9	task 10	task 11	task 12
7t- 8t	task 7	task 8	task 9	task 10	task 11	task 12
8t- 9t	task 7	task 8	task 9	task 10	task 11	task 12
9t-10t	task 7	task 8	task 9	task 10	task 11	task 12
10t-11t	task 7	task 8	task 9	task 10	task 11	task 12
11t-12t	task 7	task 8	task 9	task 10	task 11	task 12
12t-13t	task 13	task 14	task 15	task 16	task 17	task 18
etc.	¦	¦	¦	¦	¦	¦

Figure 4. Six module parallel-activated parallel array. (delay per module = 6t)

Requirements for a Dataflow Computer

The description of dataflow notation earlier in the paper showed that a dataflow computer (static or dynamic) needs to perform three major tasks:

(i) it must store a representation of the program graph;

(ii) it must implement the equivalent of data tokens which can flow through the graph and match together with appropriate partners;

and (iii) it must provide suitable instruction processing facilities.

Static and dynamic code-copying systems implement tokens by providing additional data storage space in the program graph store. Dynamic tagged systems require separate code and data stores. Instruction sets for the different schemes reflect the needs discussed earlier (e.g. apply for code-copying systems and change tag for code-sharing systems).

In terms of system architecture any of the schemes introduced in the previous section could be used. In practice systems are designed to be extensible via the addition of extra hardware modules, and so the pipeline is not attractive as an overall structure. Consequently systems under construction are based on parallel-activated parallel arrays of dataflow processing elements. Pipelines are sometimes used within the processing elements.

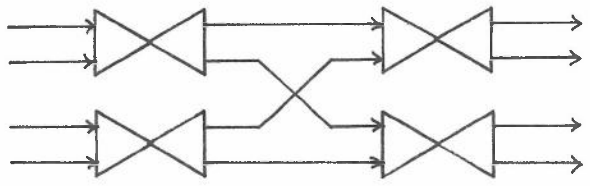
We shall first describe two systems in which conventional microprocessors are used for these processing elements. Two differently motivated refinements of this kind of system are then considered. Each system is either already in operation or currently under construction.

Static Dataflow Multiprocessors

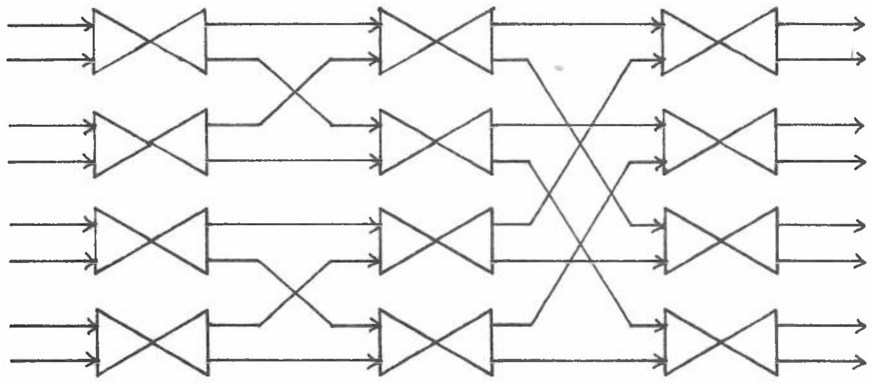
The two systems based on conventional microprocessors have structures identical to that shown in Fig.4. They differ in the microprocessor architecture used and in the nature of the communication switch. A research system built in 1978 by a team led by Don Oxley at Texas Instruments used four microengines and a 990/10 host, connected together via a time-multiplexed communication ring [15]. A more powerful system is currently being implemented by Jack Dennis and his group at MIT using specialised bit-sliced microengines connected via a general purpose unidirectional routing network [11]. In both systems the three major dataflow tasks are implemented in software in the microprocessor modules. The TI system can use a slow communication ring because of the relatively slow processing speed of its microengines and the small number of processors that need to be linked. In the MIT system the microengines are faster and there are more of them so the switch needs to have much higher throughput. Both systems implement a static dataflow scheme with possible extension to dynamic code-copying.

Code for the TI system is produced from standard Fortran programs whereas a single-assignment language (Val) has been defined and implemented to act as the high-level interface for the MIT machine [2]. Arrays are provided in both languages, but streams are also anticipated in Val.

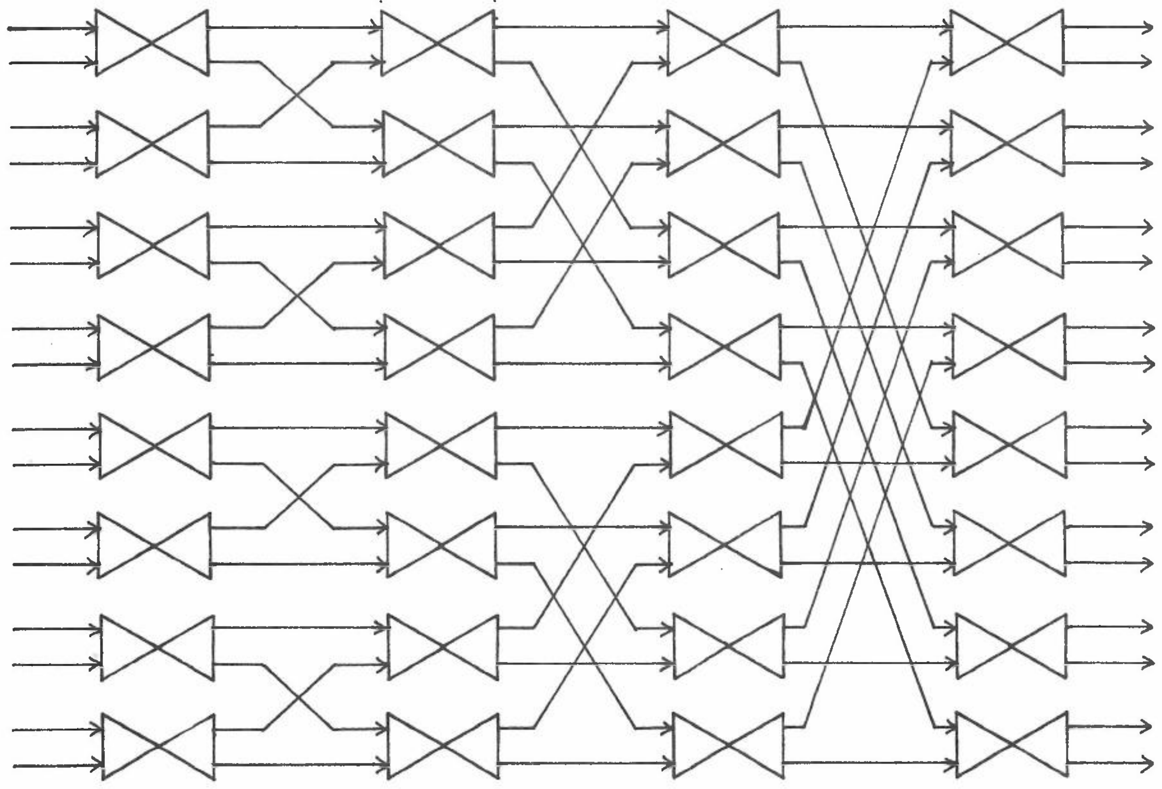
A key factor in the design of the MIT system is the ability to expand its power by adding extra processors via an extended communications switch. The switch itself is a network of 2x2 routers connected in such a way that data arriving at any input port may reach any output port in time proportional to the logarithm of the number of processors. The structure of the switch for 4, 8, and 16 processors is shown in Fig.5. Each 2x2 element contains byte-wide data paths.



(a) 4x4 network



(b) 8x8 network



(c) 16x16 network

Figure 5. Communication networks using 2x2 routers.

The desire to expand power by adding hardware is common to all dataflow system designs. The systems described below also use switches that are similar to those shown in Fig.5. There is keen debate about the maximum size of switch that can be constructed (or that will be feasible in the forseeable future). There is an obvious relationship between the power of individual processors, the total power, and the size of the switch. Because the systems described above use conventional microprocessor software to emulate the dataflow model, they run relatively slowly and large switches will be needed for substantial applications (e.g. weather forecasting). For example, it is implicit in the MIT design that switches of size 500x500 and more can be implemented using byte-wide 2x2 routers.

Other researchers are less confident that switches of this size will be practicable. Consequently they have concentrated on improving the execution rates of individual processors by designing their hardware to be dataflow oriented. The projects described below follow this approach, and also both implement dynamic tagged schemas.

An MSI Dynamic Tagged Dataflow Processor

A research group at Manchester University under the leadership of Ian Watson and the author has constructed a specialised 'ring-structured' dataflow processor with funding from the Distributed Computing Systems Programme of the Science and Engineering Research Council of Great Britain [13]. In this ring-structure the three dataflow tasks (i.e. matching tokens together; finding the next instruction; and processing of instructions) are implemented in three separate hardware modules. The individual actions in these modules are dependent solely on the module input data so that successive actions may be overlapped by connecting the modules in a pipeline. One extra pipeline module is provided to queue excess tasks when highly parallel programs are running. The overall ring-structure is therefore a four-stage pipeline as shown in Fig.6.

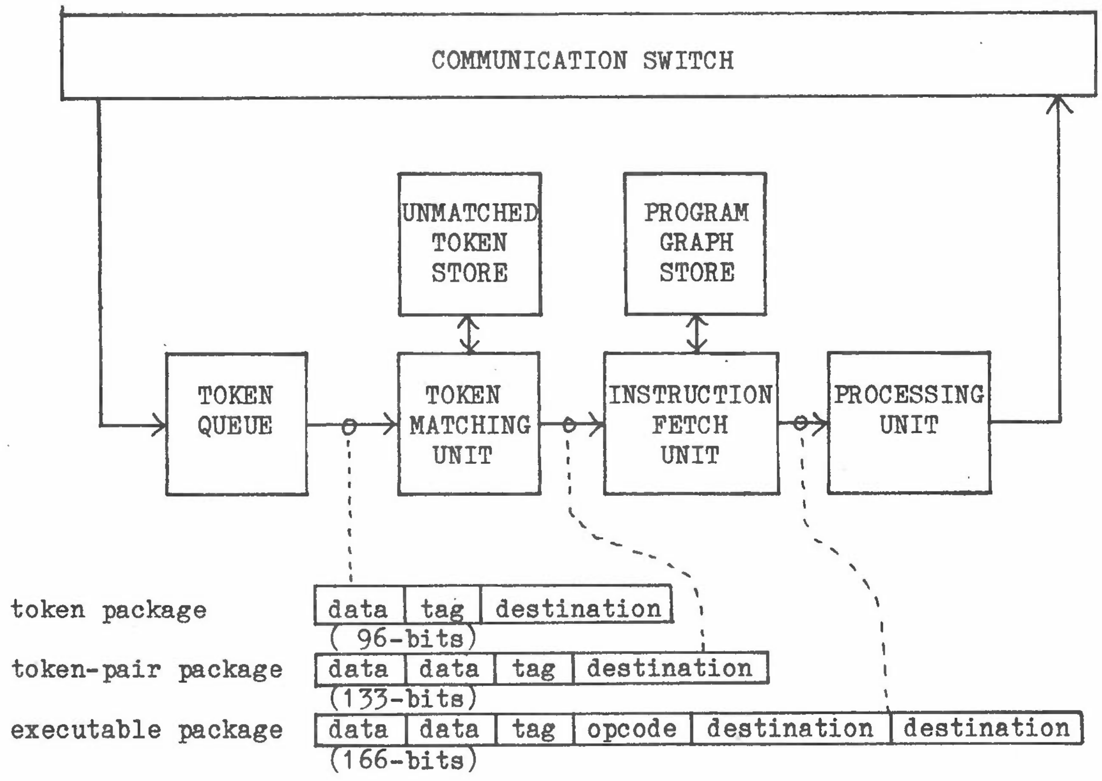


Figure 6. Ring-structured dataflow processor.

The fundamental unit of data in the switch is a token-package representing a notational tagged token on an arc of the program graph. The token has a data type and value and a tag. The arc is represented by the address (in the program graph store) of the instruction at its head (known as the 'destination'). The token is the smallest data package in the system, and so the queue module is positioned adjacent to the switch, at the input to the ring.

Queued tokens are presented one-at-a-time to the matching unit, which is responsible for grouping together tokens with the same tag heading for the same destination instruction. In the Manchester system tokens may be grouped together in ones or twos so that triadic instructions and above cannot be supported. Tokens which expect to find a partner, but which arrive at the matching unit before the partner does, are kept in the unmatched-token store until the partner arrives. At this time (or, in the case of a single-input instruction, when the first and only token arrives) all the required input data and the common tag and destination fields are sent to the instruction fetch unit as a token-pair package.

The program graph is stored as an array of instructions each representing one operator and its associated output arc(s). The destination field of an incoming token-pair is used as an address to fetch the next instruction which contains an opcode and up to two destination fields. This produces a complete executable package which is sent to the processing unit. Here the specified opcode is executed using the collected data and tag as operands, and the result token(s) is(are) finally returned to the communications switch input.

The critical part of this system is the matching unit. The task of pairing tokens together is an act of association and so the unmatched-token store is (pseudo-)associative in nature. Details of the operation of the matching unit are beyond the scope of the present paper. In the technology chosen for the prototype version the average match time is 300 nanoseconds [19]. This limits the instruction execution rate of the ring-structured processor to 3.3 million instructions per second (MIPS). The prototype instruction processing element is some twenty times slower than this and so a serially-activated parallel array of 20 such elements is used as a processing unit. At the time of writing (January 1983) the prototype system is running at just less than 2MIPS with 12 elements in the array.

The prototype implementation is tailored to stable, MSI, medium-speed, TTL technology. Higher speed could be obtained using faster logic and storage components, for example ECL. Comparable speed might be obtained if design were tailored to VLSI technology. The next section describes a project in which this latter approach is being followed.

A VLSI Dynamic Tagged Dataflow Processor

Another research group at MIT, under the leadership of Arvind, is constructing a VLSI-based dataflow processor with many of the characteristics of the ring-structured system. The main differences between this system and the MSI-based implementation are that (i) data structure accesses are handled separately from ordinary token activities, and (ii) there is a two-tiered communication system [5]. The processor design is outlined in Fig.7.

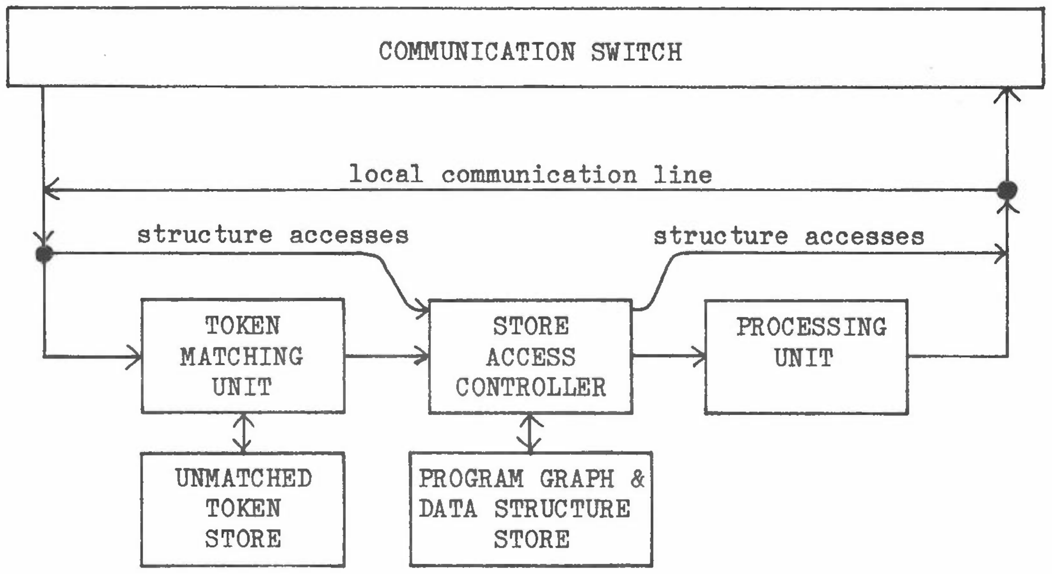


Figure 7. VLSI-based dataflow processor.

Data structure operations are treated separately so that (i) they can be performed quickly, and (ii) the potentially large numbers of tokens involved do not occupy space in the expensive unmatched-token store. The two-tiered communication structure relieves the general communications switch of excess traffic as long as programs exhibit strong 'locality' (i.e. processing activity is localised in subgraphs and processors rather than communicating randomly with other subgraphs/processors). Locality also benefits the size of the unmatched-token store, and current plans at MIT are to implement a small 64-word store instead of the 16k-word version in use at Manchester.

Reduced traffic in the communications switch allows bit-wide data paths to be used. The proposed building block for this MIT system is an 8x8 bit-wide module. Using program locality still further, large-size switches can be made rather less complex than the networks shown in Fig.5.

This design relies heavily on strong program locality. The language Id [4], also developed by Arvind's team, has appropriate properties, and the system is being designed around this language.

Summary of Dataflow Hardware

The great advantage of dataflow notation is that individual actions in executing a graph program interact with one another solely by token flow along the dependence arcs. All necessary synchronisation of data and instructions is achieved by the act of 'matching' tokens together. Hence, whether pipelines or parallel arrays are used for implementation, the parallel hardware can be kept occupied by simple distribution techniques without having to worry about inconvenient interlinkings of constituent modules.

The systems presented above have similar overall structure but there are considerable differences in detail. These are by no means the only system designs in existence. Three other projects are worthy of mention. Two of these have had operational hardware for several years. They are the LAU project led by Jean-Claude Syre at CERT Toulouse in France [9], and the DDM1 project led by Al Davis at the University of Utah [10]. The third system is being developed at NTT in Japan by Makoto Amamiya and his group [3]. Their hardware has been operational for about one year. Additional information on dataflow systems and languages has recently been published in a special issue of IEEE Computer [1].

Little has yet been published about the performance of these dataflow systems in practical applications. The existence of so much operational hardware should lead to an expansion of assessment work, and subsequent results, in the near future. The Japanese have stated their intention to discover the abilities of such systems within three to five years. We anticipate a similar timescale for study in Europe and the USA.

References

[1] IEEE Computer, vol.15, no.2
Special Issue on Data Flow Systems
February 1982

[2] Ackerman W B and Dennis J B
VAL - A Value-Oriented Algorithmic Language: Preliminary Reference Manual
MIT Laboratory for Computer Science
Technical Report TR-218
June 1979

[3] Amamiya M, Takahashi N, Naruse T and Yoshida M
A Data Flow Processor Array System for Solving Partial Differential Equations
Proc. International Symposium on Applied Mathematics and Information Science
Kyoto University
March 1982

[4] Arvind, Gostelow K P and Plouffe W
An Asynchronous Programming Language and Computing Machine
University of California at Irvine
Dept. of Information and Computer Science
Technical Report TR114a
December 1978

[5] Arvind and Kathail V
A Multiple Processor Dataflow Machine That Supports Generalised Procedures
Proc. 8th Annual ACM Symposium on Computer Architecture
May 1981, p291

[6] Barnes G H, Brown R M, Kato M, Kuck D J, Slotnick D L and Stokes R A
The ILLIAC IV Computer
IEEE Transactions on Computers, vol.C-17, no.8
August 1968, p746

[7] Brinch Hansen P
The Programming Language Concurrent Pascal
IEEE Transactions on Software Engineering, vol.SE-1, no.2
June 1975

[8] Brinch Hansen P
Distributed Processes : A Concurrent Programming Concept
Communications of the ACM, vol.21, no.11
November 1978, p934

[9] Comte D, Hifdi N and Syre J C
The Data Driven LAU Multiprocessor System: Results and Perspectives
Proc. IFIP 80 Congress
October 1980, p175

[10] Davis A L
The Architecture and System Methodology of DDM1: A Recursively Structured Data Driven Machine
Proc. 5th Annual ACM Symposium on Computer Architecture
April 1978, p210

[11] Dennis J B, Boughton G A and Leung C K C
Building Blocks for Data Flow Prototypes
Proc. 7th Annual ACM Symposium on Computer Architecture
May 1980, p1

[12] Flynn M J
Some Computer Organisations and Their Effectiveness
IEEE Transactions on Computers, vol.C-21, no.9
September 1972, p948

[13] Gurd J R and Watson I
Dataflow System for High Speed Parallel Computing
Computer Design, vol.19, no.6 & no.7
June 1980, p91 & July 1980, p97

[14] Hoare C A R
Communicating Sequential Processes
Communications of the ACM, vol.21, no.8
August 1978, p666

[15] Johnson D
Automatic Partitioning of Programs in Multiprocessor Systems
Proc. IEEE Spring COMPCON
April 1980

[16] Reddaway S F
DAP - A Distributed Array Processor
Proc. 1st Annual ACM Symposium on Computer Architecture
December 1973

[17] Swan R J, Fuller S H and Siewiorek D P
Cm* - A Modular Multimicroprocessor
Proc. AFIPS NCC, vol.46
June 1977, p637

[18] Thornton J E
Design of a Computer: The CD6600
Scott, Foresman & Co.
1970

[19] Watson I and Gurd J R
A Practical Dataflow Computer
IEEE Computer, vol.15, no.2
February 1982, p51

[20] Wirth N
Modula: A Language for Modular Multiprogramming
Software - Practice & Experience, vol.7, no.1
January 1977, p3

[21] Wulf W A and Bell C G
C.mmp - A multi-mini-processor
Proc. AFIPS FJCC, vol.41
September 1972, p765

THE MANCHESTER DATAFLOW COMPUTER

1. BRIEF SUMMARY OF DATAFLOW TECHNIQUES
2. THE MANCHESTER PROTOTYPE - HARDWARE AND SOFTWARE
3. EVALUATION METHOD - MEASUREMENTS AND MEANING
4. EVALUATION RESULTS
5. INTERPRETATION AND FUTURE WORK

DATAFLOW PROGRAM GRAPHS

INSTRUCTIONS APPEAR HORIZONTALLY IF THEY CAN BE EXECUTED CONCURRENTLY, AND VERTICALLY IF THEY MUST BE EXECUTED IN SEQUENCE.

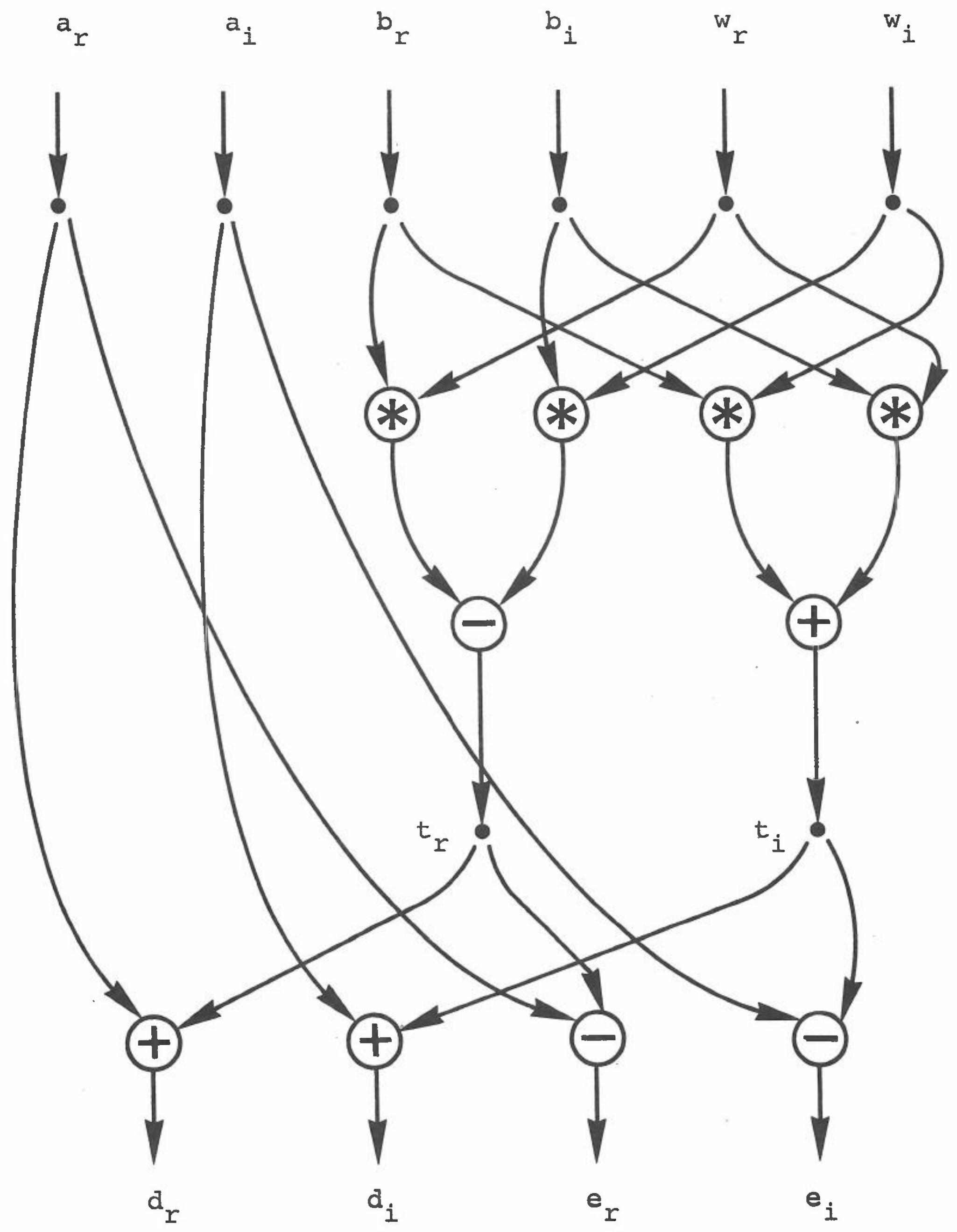
SEQUENCING CONSTRAINTS ARE INDICATED BY DATA DEPENDENCE ARCS DRAWN BETWEEN THE INSTRUCTIONS.

INSTRUCTIONS DO NOT REFERENCE MEMORY - DATA (IN THE FORM OF TOKEN PACKETS) IS SENT DIRECTLY ALONG THE DATA DEPENDENCE ARCS.

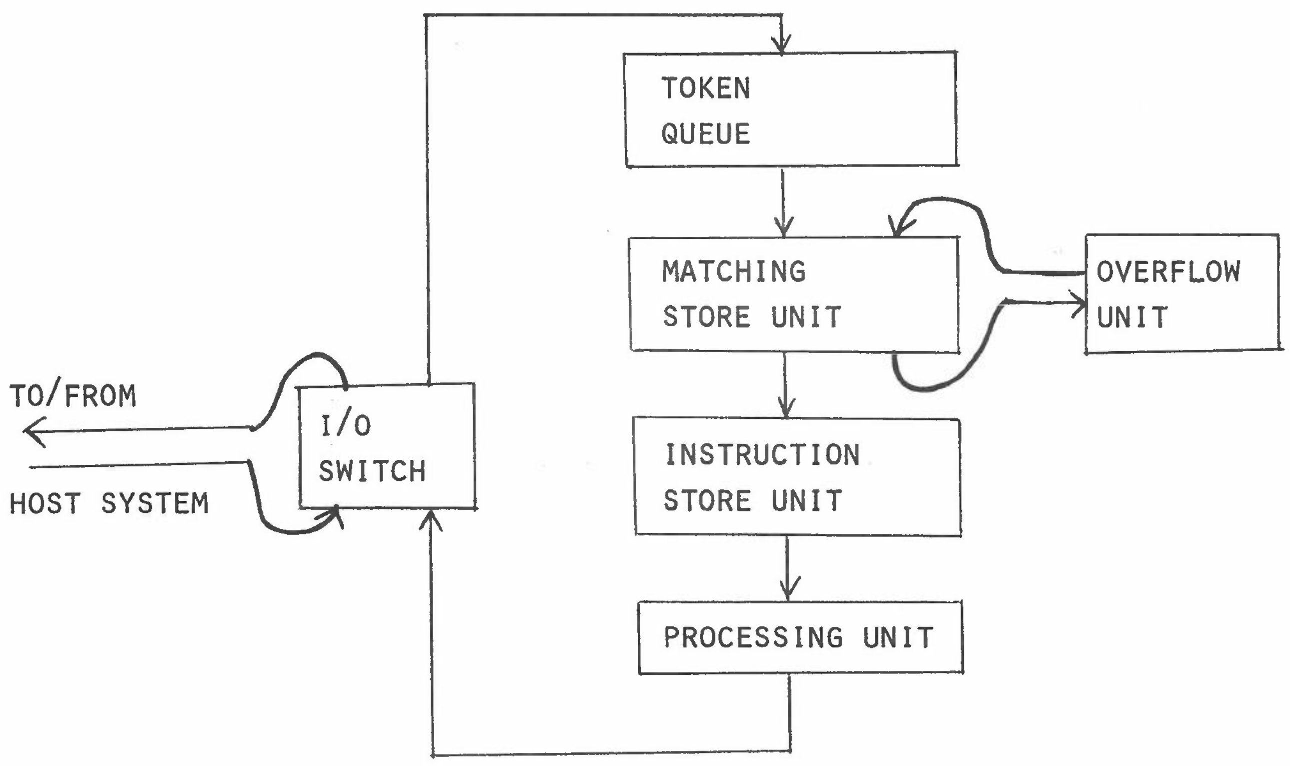
INSTRUCTION EXECUTION IS PROMPTED BY THE PRESENCE OF DATA TOKENS AT ALL INPUT POINTS.

AN EXAMPLE DATAFLOW GRAPH -

THE FFT BUTTERFLY

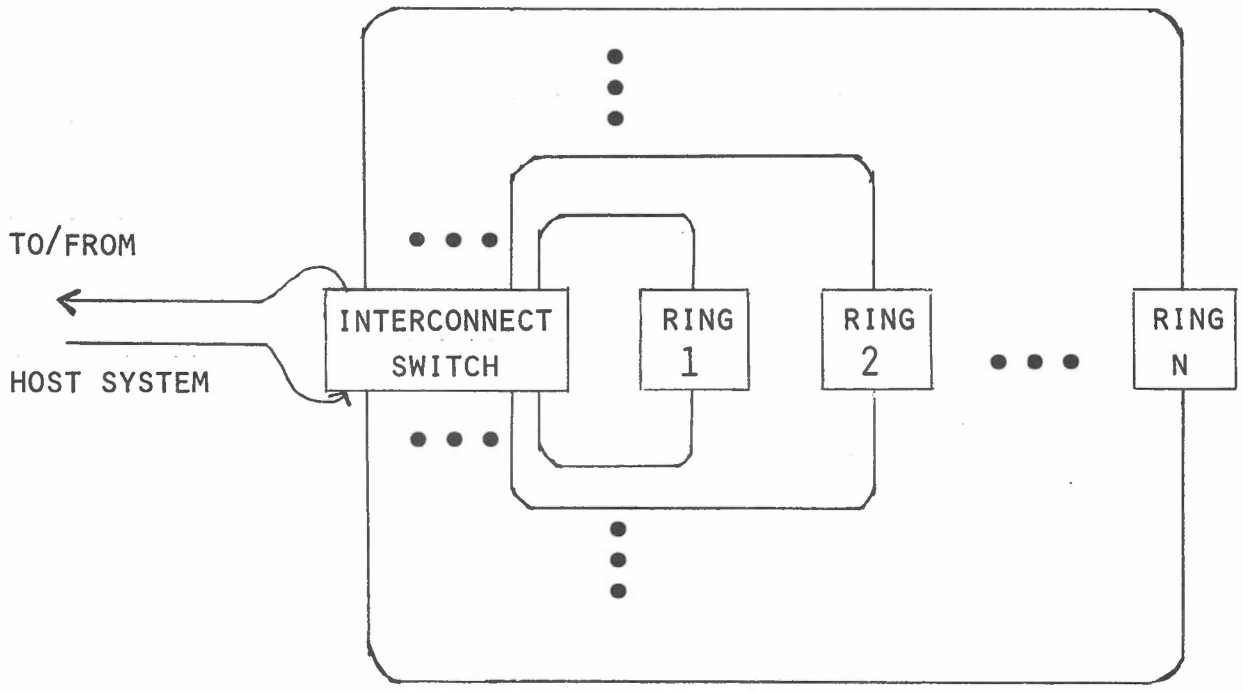


A SINGLE-RING DATAFLOW PROCESSOR

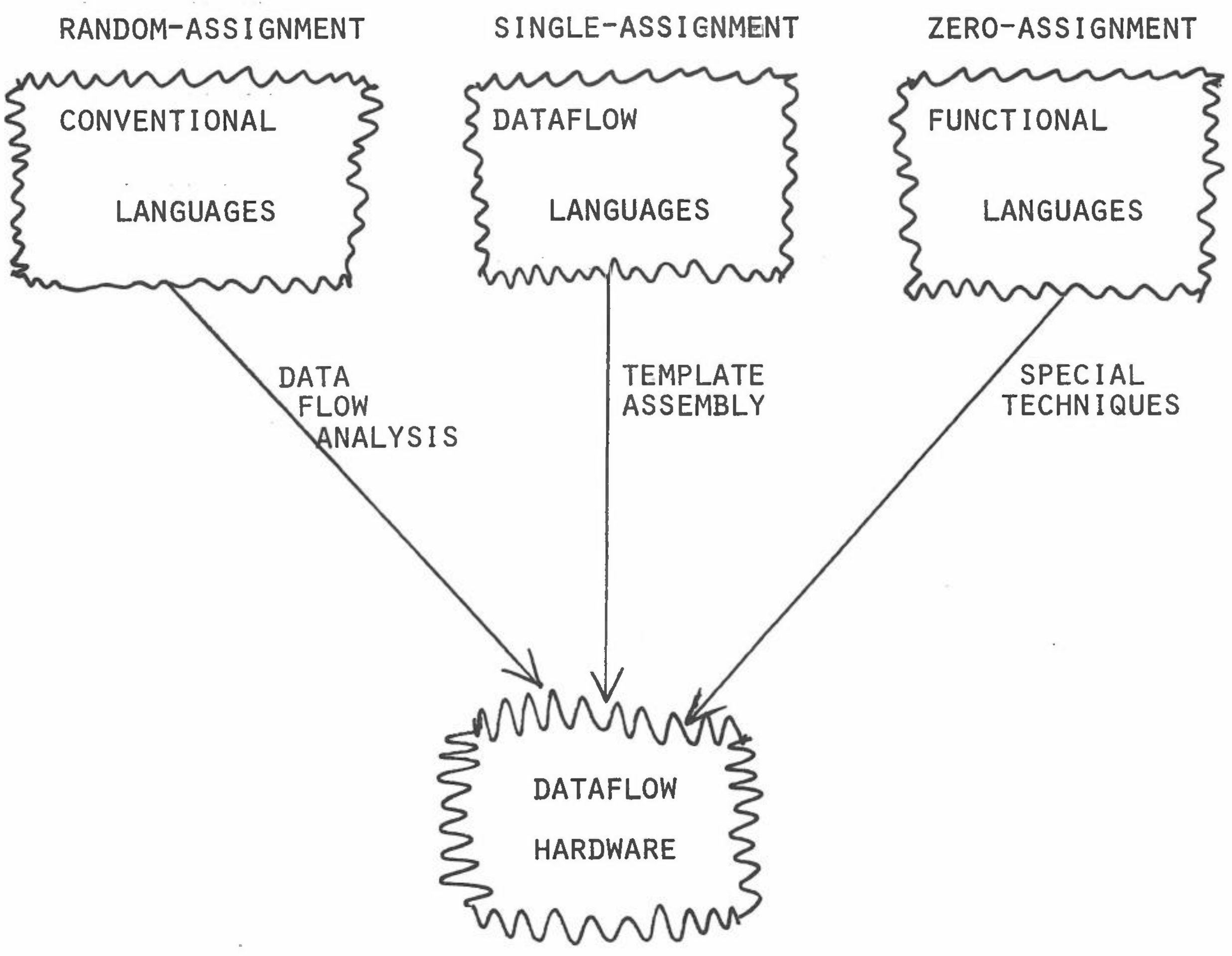


A MULTI-RING DATAFLOW PROCESSOR

GIVES 'ADD-ON' PROCESSING POWER



DATAFLOW PROGRAMMING ROUTES



ACHIEVEMENTS AT MANCHESTER,

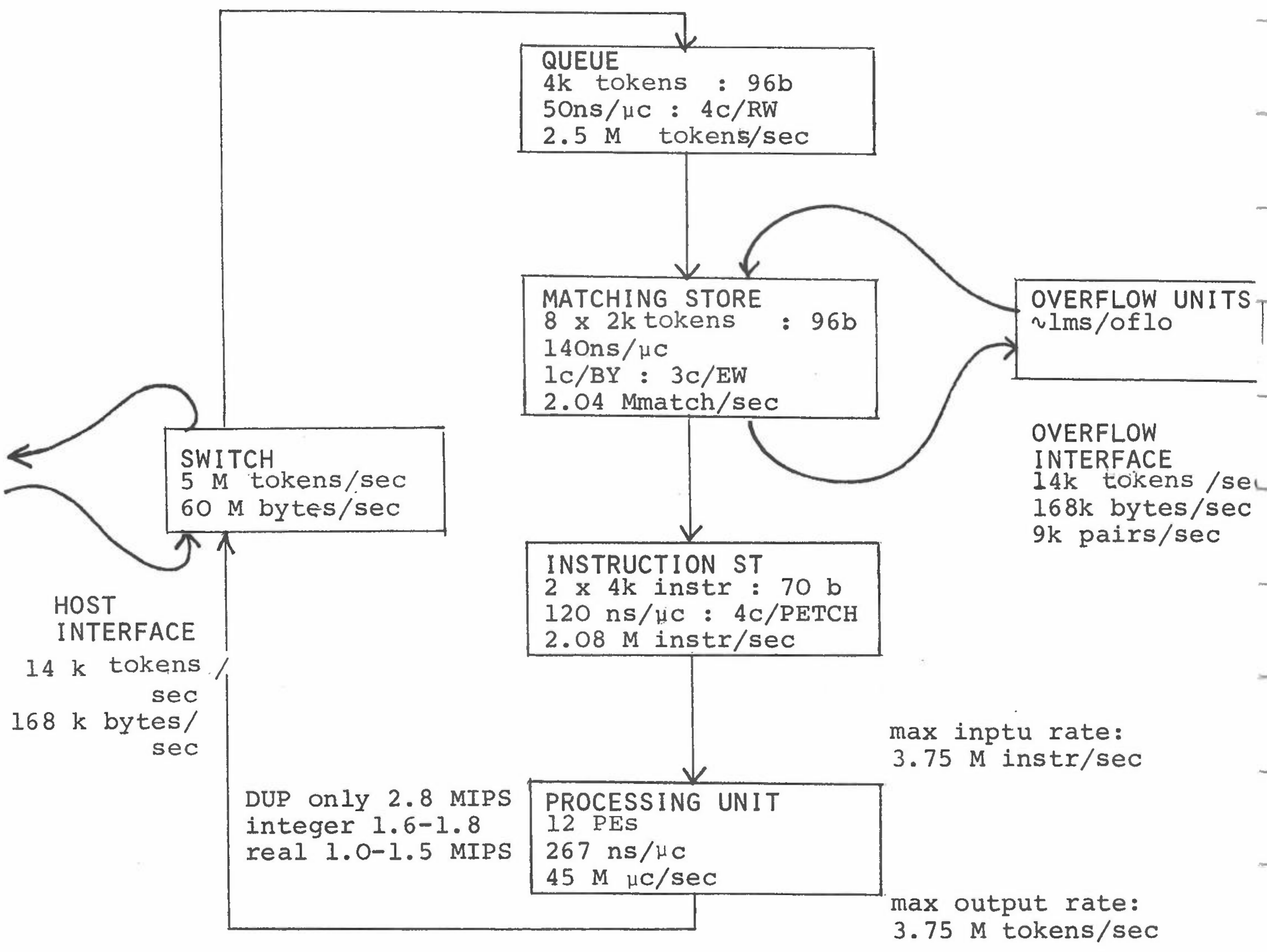
OVERVIEW

1. PROTOTYPE DATAFLOW HARDWARE RUNNING 1 - 1.5 MIPS

2. PROTOTYPE SOFTWARE SUPPORT SYSTEM IN OPERATION

3. PRELIMINARY PERFORMANCE EVALUATION IN PROGRESS

MANCHESTER DATAFLOW HARDWARE CURRENT STATUS

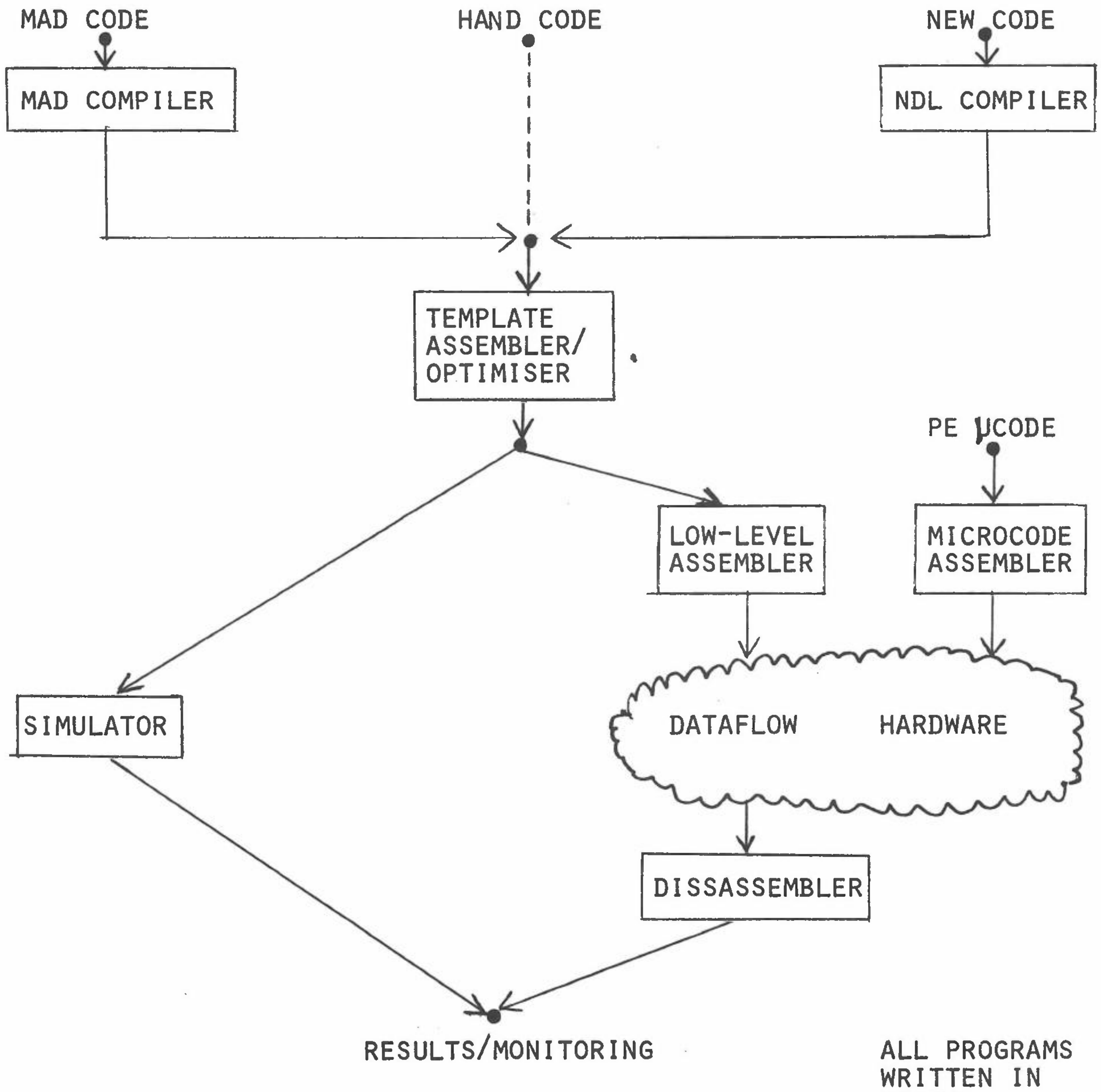


* is average rate - all others maxima

∿ 4000 debugged MSI TTL ICs

90% of opcodes implemented (equivalent to simulator)

THE SOFTWARE ENVIRONMENT



ALL PROGRAMS
WRITTEN IN
PASCAL

EVALUATION OBJECTIVES

1. TUNE PROTOTYPE HARDWARE FOR OPTIMUM PERFORMANCE.
2. DETERMINE THE NATURE OF SOFTWARE PARALLELISM THAT CAN BE EXPLOITED BY THE HARDWARE.
3. ESTABLISH THE VALUE OF A DATAFLOW MIP.

EVALUATION METHOD

1. FOR PROGRAMS THAT DO NOT OVERFLOW

   * PLOT SPEEDUP CURVES
   * INTERPRET RESULTS
   X RECTIFY PROBLEMS IN PROCESSOR AND PIPELINE HARDWARE

2. FOR PROGRAMS THAT OVERFLOW A LITTLE

   DETERMINE NATURE OF BOTTLENECKS IN OVERFLOW LOOP
   RECTIFY PROBLEMS IN HARDWARE/SOFTWARE

3. FOR PROGRAMS THAT OVERFLOW A LOT

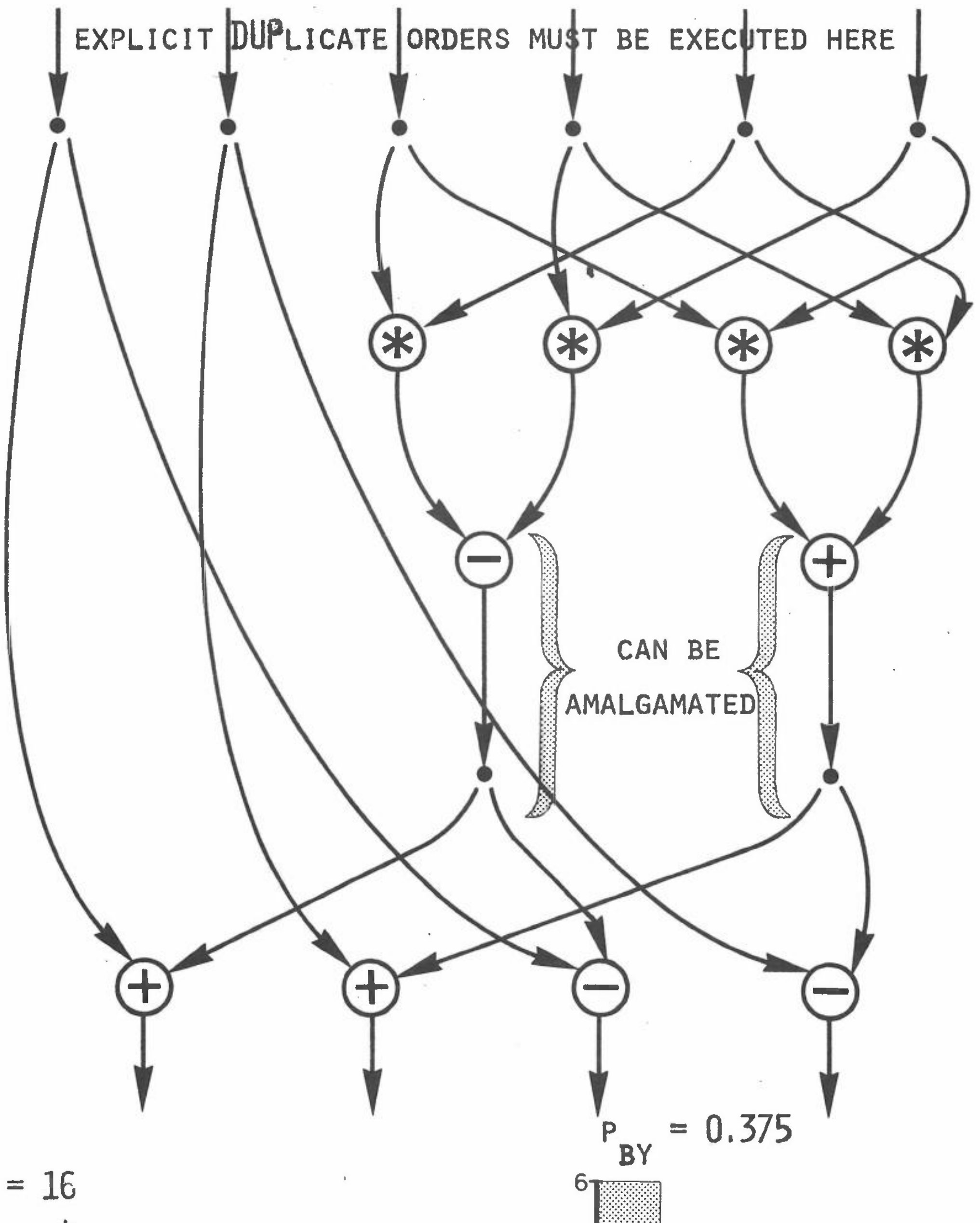
   DESIGN AND IMPLEMENT A HIERARCHICAL MEMORY
   EVALUATE AND OPTIMISE PERFORMANCE

BASIC PROGRAM MEASUREMENTS

S_1	TOTAL NUMBER OF INSTRUCTIONS EXECUTED $\equiv$ NUMBER OF EXECUTION STEPS WITH 1 PE
S_∞	NUMBER OF EXECUTION STEPS WITH UNLIMITED PEs (N.B. NOTIONAL ONLY: STEPS ARE NOT OF EQUAL TIME)
$\Pi = \frac{S_1}{S_\infty}$	A ROUGH MEASURE OF PROGRAM PARALLELISM (BUT DOES NOT ACCOUNT FOR TIME-VARIANCE OF PARALLELISM)
P_{BY}	PROPORTION OF EXECUTED INSTRUCTIONS WHICH HAVE ONE INPUT (I.E. BYPASS MATCHING FUNCTION)

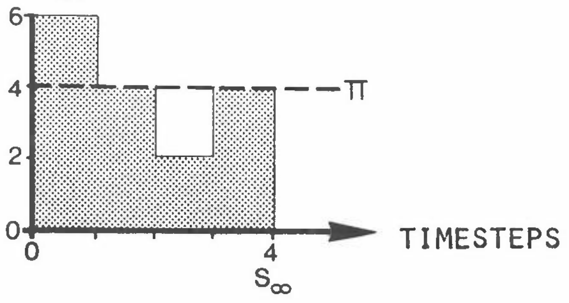
EXAMPLE PROGRAM MEASUREMENTS

THE FFT BUTTERFLY



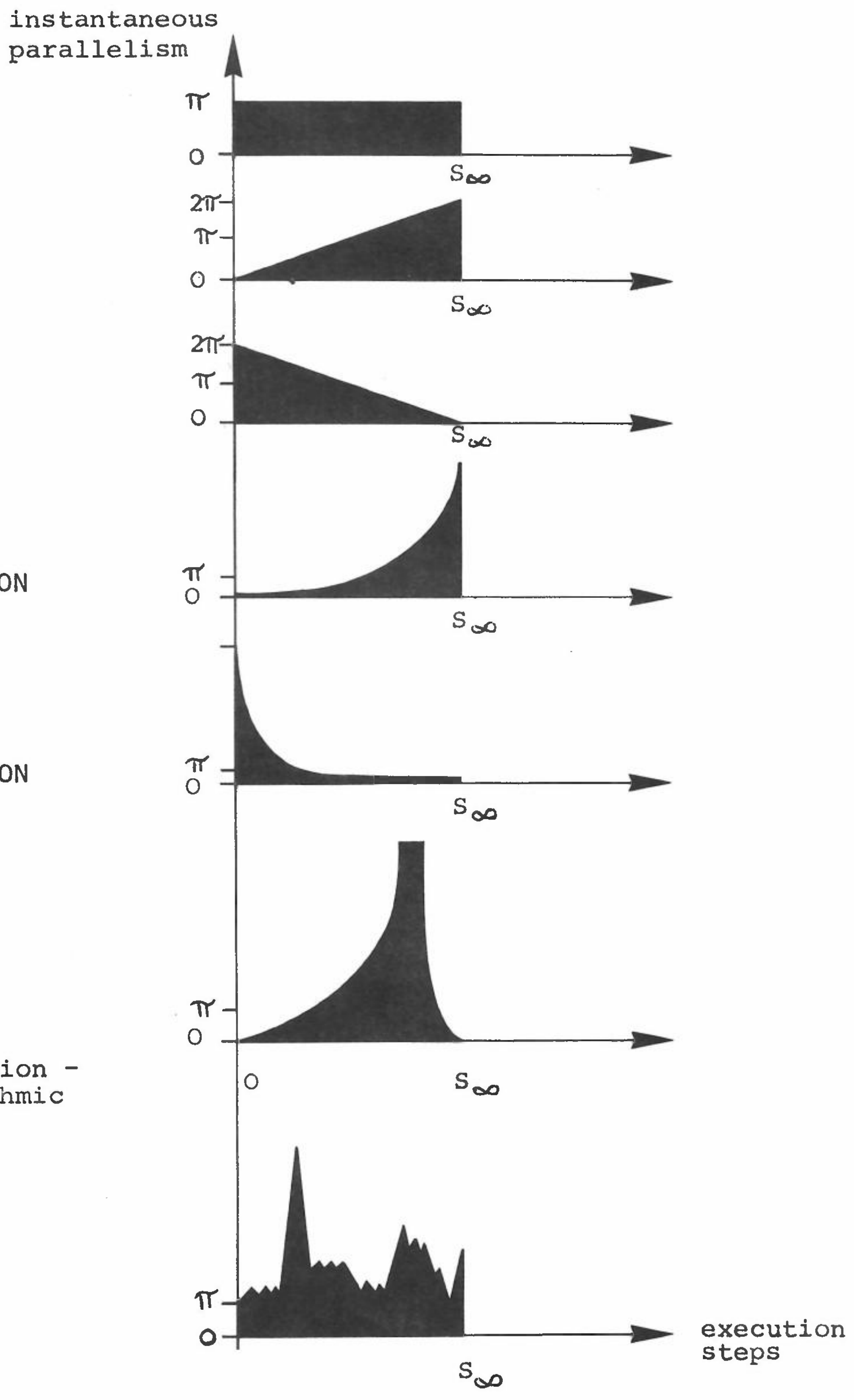
$s_1 = 16$
$s_\infty = 4$
$\pi = 4$

$P_{BY} = 0.375$



TIME-VARIANCE OF PROGRAM PARALLELISM

(AREA OF EACH BLOCK = S_1)



1. CONSTANT

2. LINEAR EXPANSION

3. LINEAR REDUCTION

4. EXPONENTIAL EXPANSION

5. LOGARITHMIC REDUCTION

6. RECURSIVE DOUBLING
(exponential expansion - constant - logarithmic reduction)

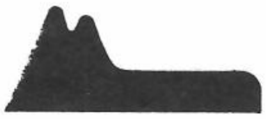
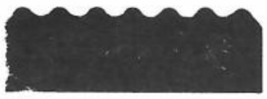
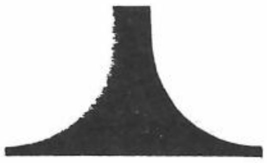
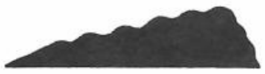
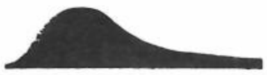
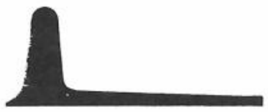
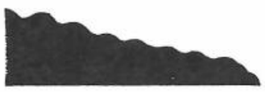
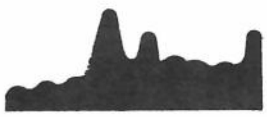
7. IRREGULAR
(mixed)

BASIC HARDWARE MEASUREMENTS

T_1	TOTAL EXECUTION TIME FOR 1 PE
T_n	TOTAL EXECUTION TIME FOR n PEs
$P_n = \frac{T_1}{T_n}$	EFFECTIVE NUMBER OF PEs WHEN n ARE ACTIVE
$E_n = 100 \frac{P_n}{n}$	% UTILISATION OF n PEs
$M_n' = n \frac{S_1}{T_1}$	POTENTIAL MIP RATE FOR n PEs
$M_n = \frac{S_1}{T_n}$	ACTUAL MIP RATE FOR n PEs

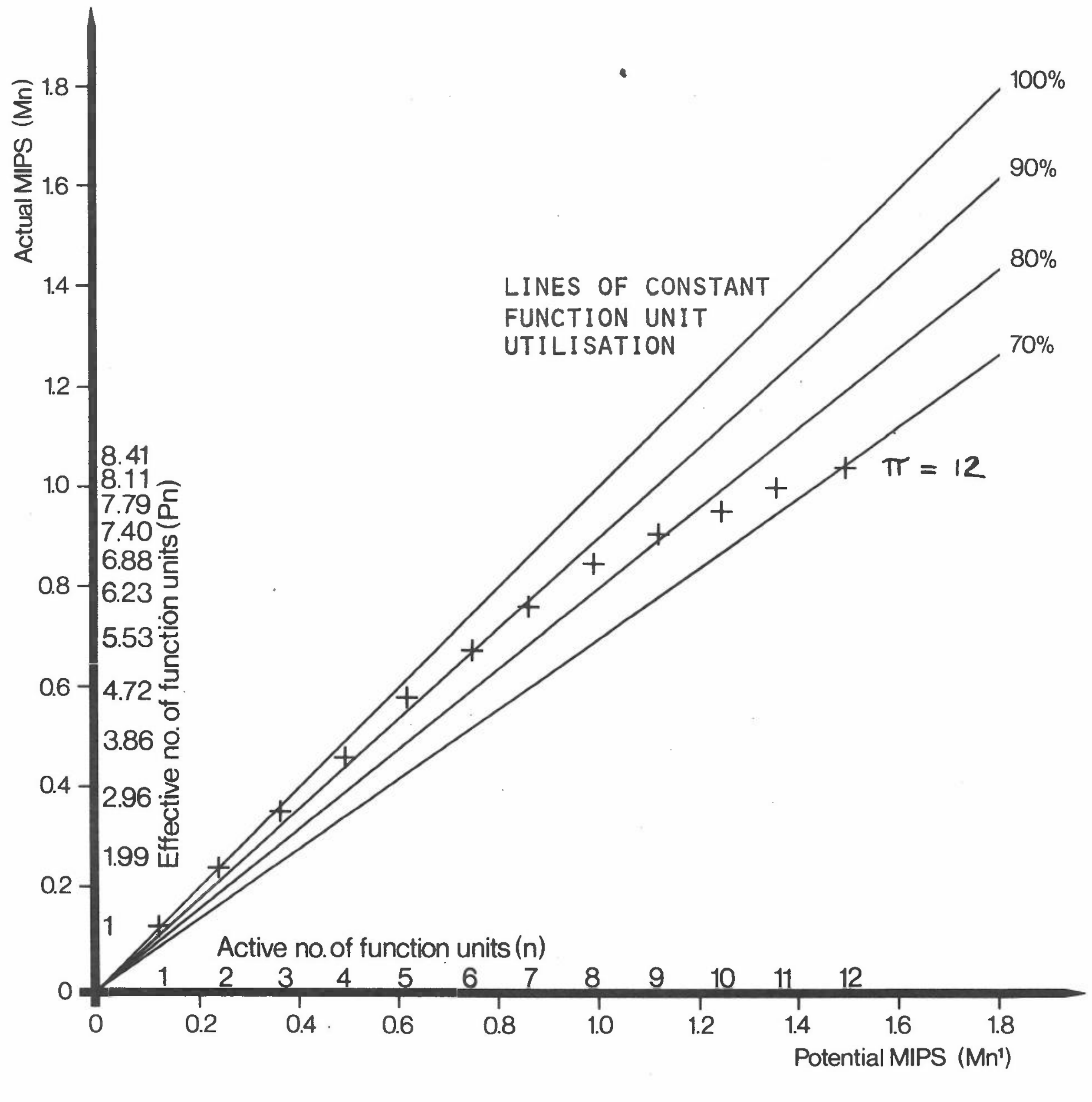
BENCHMARK CODES

(SIMPLE AND NO OVERFLOWS GENERATED)

NAME	SOURCE	PARALLELISM		P_{BY}	COMMENT
FFT	MACRO		$\pi = 50/100$	.7	COMPLEX 32/64 POINT FFT
LAPLAC	MACRO ASSEM		$\pi = 50/130$	.7	REAL ITERATIVE LAPLACE RELAXATION
SUMPRO	MAD		$\pi = 2 \rightarrow 150 +$	.61	RECURSIVE DOUBLING INTEGER SUM
INTC	MAD		$\pi = 12$	.64	TRAPEZOIDAL INTEGRATION
PLUM1	MACRO ASSEM		$\pi = 50$	.61	PLUMBLINE ALGORITHM
PLUM2	MAD		$\pi = 20$	.56	
HUGE	MACRO ASSEM		$\pi = 50/70$	.63	LOGIC SIMULATION
LEETC	MAD		$\pi = 20/40$	.61	LEE ROUTER TEST

SPEEDUP CURVE

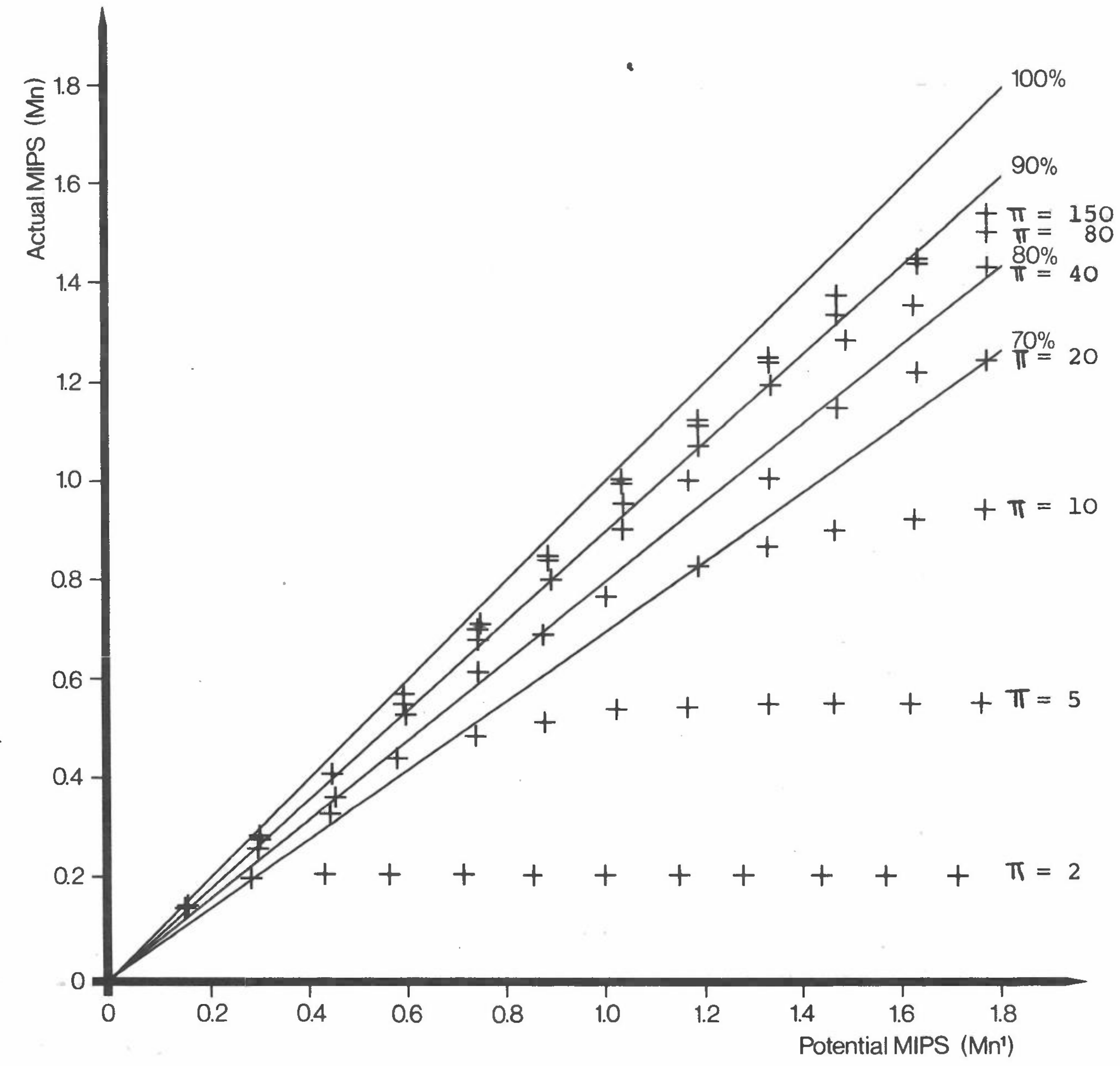
+ INTC $\pi = 12$ $P_{BY} = 0.64$



SPEEDUP CURVES

EFFECT OF VARIABLE π

+ SUMPRO π FROM 2 TO 150 $P_{BY} = 0.61$

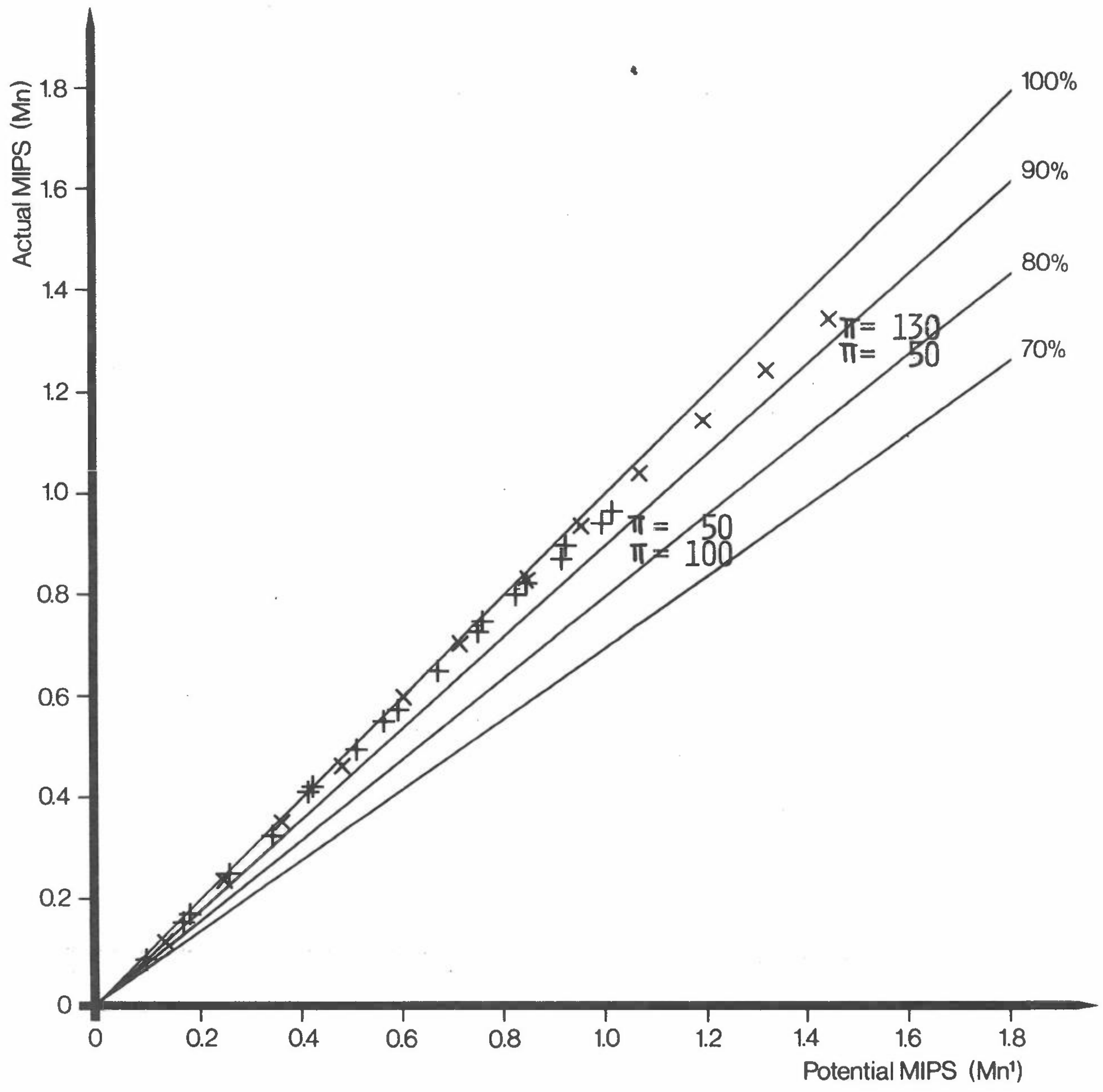


SPEEDUP CURVES

EFFECT OF VARIABLE INSTR. MIX

+ FFT $\pi = 50 / 100$ $P_{BY} = 0.7$

X LAPLAC $\pi = 50 / 130$ $P_{BY} = 0.7$

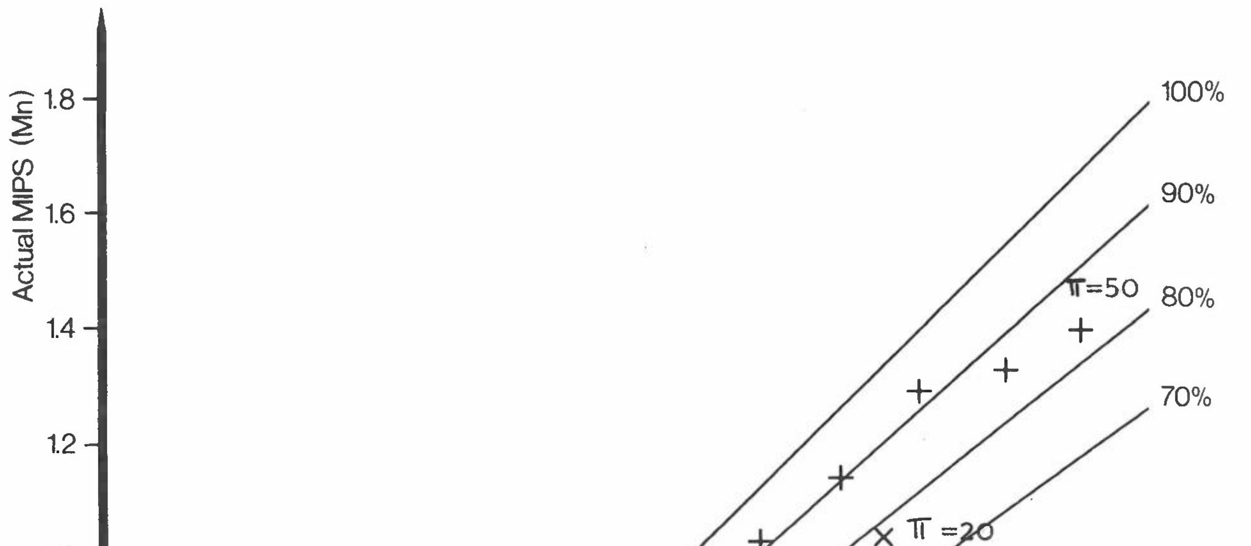


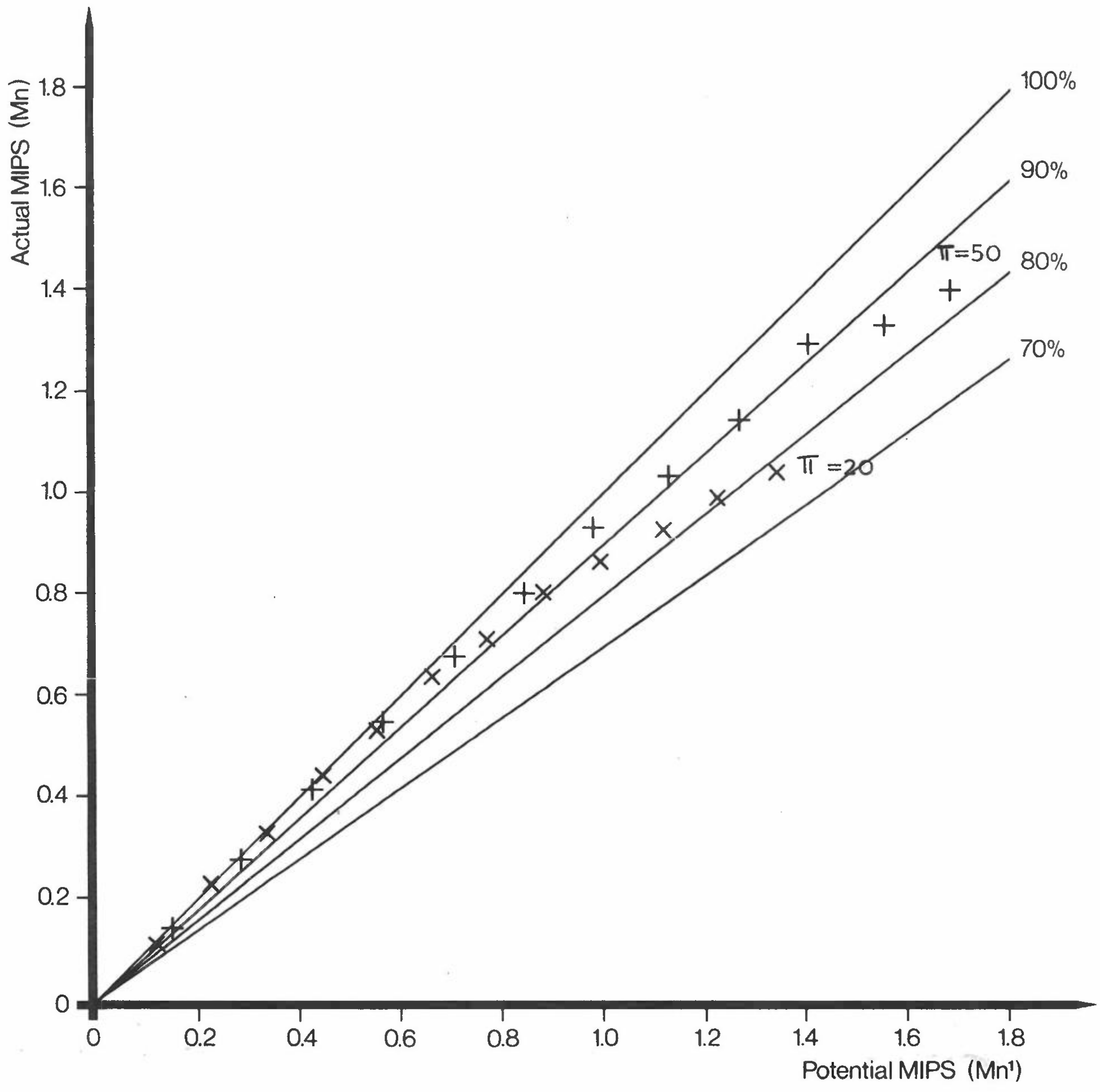
SPEEDUP CURVES

EFFECT OF SOURCE LANGUAGE

+ PLUM1 MACRO $\pi = 50$ $P_{BY} = 0.61$

X PLUM2 MAD $\pi = 20$ $P_{BY} = 0.56$



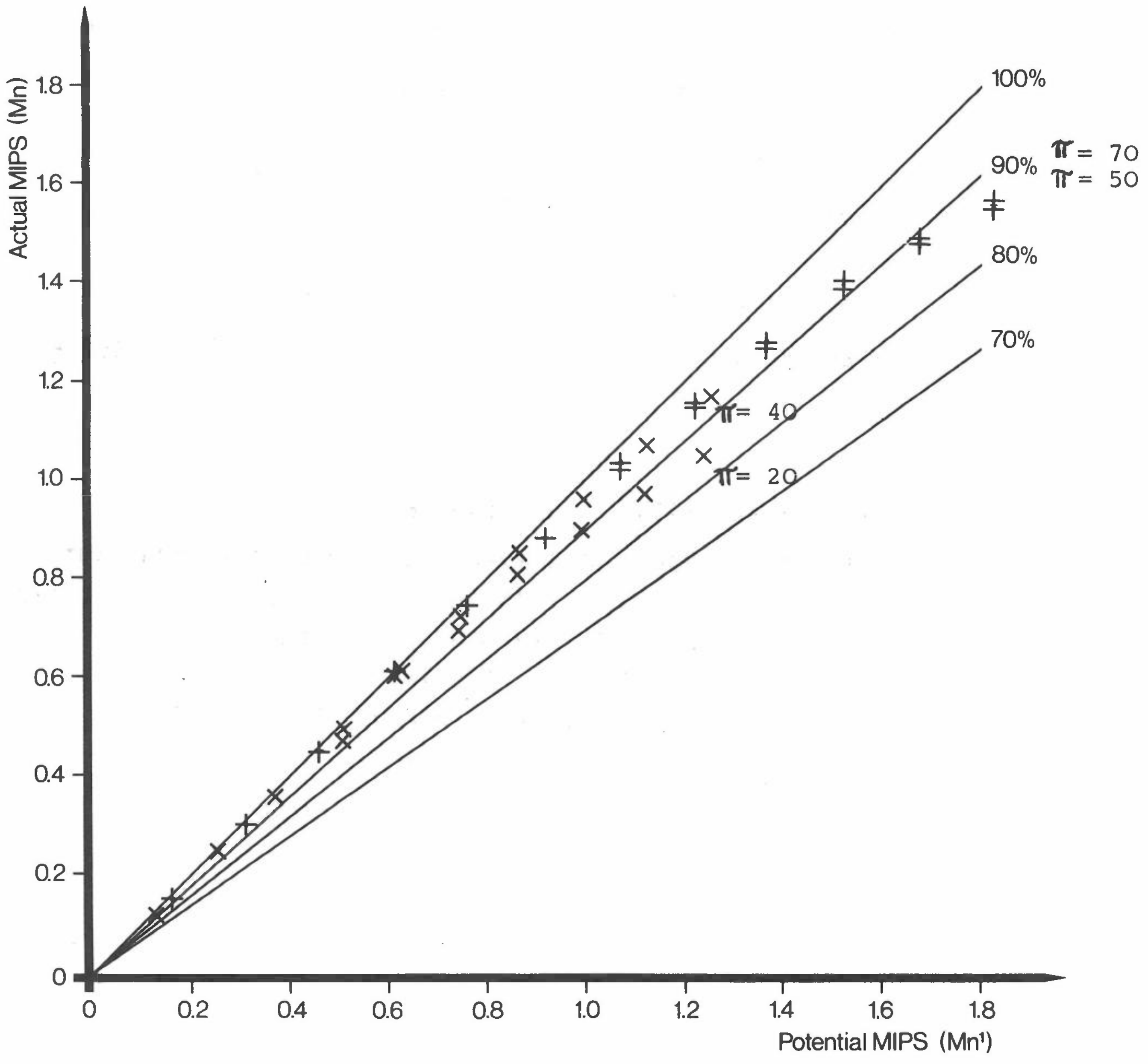


SPEEDUP CURVES

PROTOTYPE CAD CODES

+ HUGE π = 50 / 70

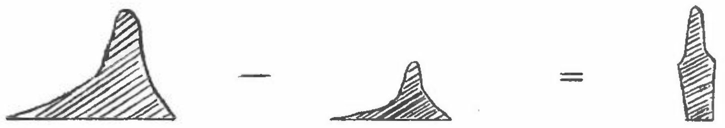
X LEETC π = 20 / 40



POSSIBLE CAUSES OF FALL-OFF IN SPEED-UP CURVES FOR HIGH VALUES OF π (WHY IS THERE A "FORBIDDEN" AREA OF HIGH UTILISATION?)

1. SERIAL SECTIONS OF CODE DOMINATE TIMING

   EXPERIMENT: ESTIMATE SERIAL SECTIONS - E.G. IN SUMPRO



2. PE OUTPUT ARBITRATOR SUFFERS CONTENTION

   EXPERIMENT: LOOK FOR EVIDENCE THAT ADDING PEs CAUSES <u>FALL</u> IN MIP RATE

3. PIPELINE BUFFERING IS INADEQUATE

   EXPERIMENT: FORCE UNIT EXECUTION TIMES TO BE CONSTANT, ESPECIALLY IN THE MATCHING UNIT - USE BYPASS AND GET VERY HIGH EXECUTION RATES SO AS TO "FLOOD" THE PIPELINE, THEN LOOK FOR SIGNS THAT FORBIDDEN AREA HAS DISSAPPEARED!

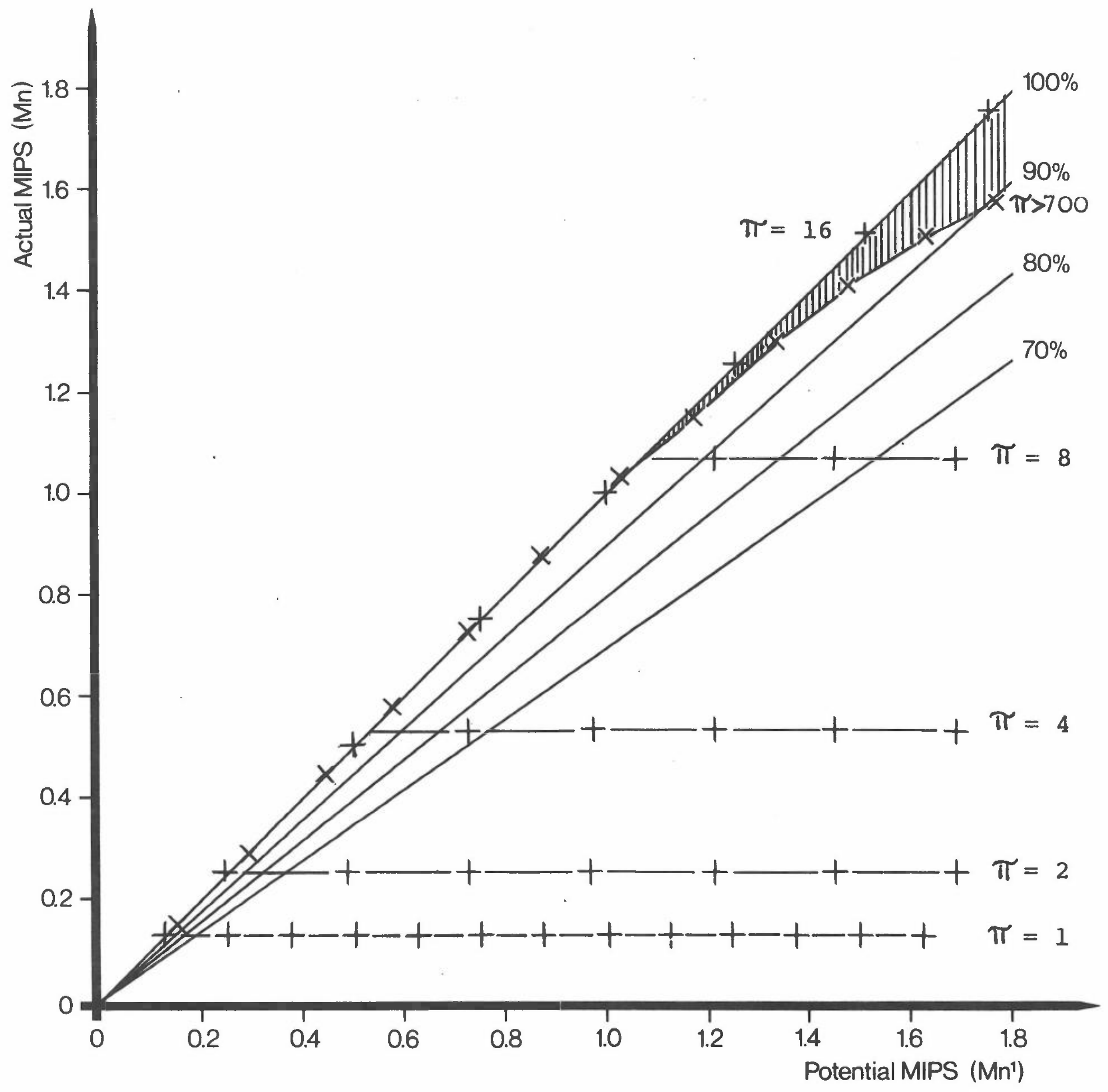
SPEEDUP CURVES

PE CONTENTION TEST / MS BYPASS TEST

+ IPCPRO $\pi = 1,2,4,8,16$ $P_{BY} = 1$

SOFTWARE PARALLELISM TEST (SIMULATED)

X SUMPRO $\pi > 700$ $P_{BY} = 0.61$



UTILISATION OF MATCHING STORE

PRELIMINARY COMMENTS

1. PRESENT HASH ALGORITHM NOT VERY EFFECTIVE, (MAY NOT EVEN BE WHAT WE THINK IT IS!)

2. OVERFLOW INTERFACE AND PROCESSING RATE FAR TOO SLOW AND UNSOPHISTICATED TO OBTAIN REALISTIC RESULTS FOR PROGRAMS WITH OVERFLOW. IMPOSSIBLE TO STUDY MEMORY HIERARCHY WITH EXISTING EQUIPMENT.

3. LARGER MATCHING STORE WILL DEFER OVERFLOW PROBLEM, BUT WILL ALSO DEFER STUDY OF MEMORY HIERARCHY.

SUMMARY

PRELIMINARY RESULTS OF EVALUATION STUDY

1. THERE IS SPEED-UP (VERSUS NUMBER OF PEs) FOR A WIDE VARIETY OF PROGRAMS: I.E. PROGRAMS HAVE PARALLELISM.

2. THE CRUDE MEASURE $\pi = \dfrac{S_1}{S_\infty}$ IS A GOOD INDICATOR OF A PROGRAM'S SUITABILITY FOR THE SYSTEM, REGARDLESS OF ANY TIME VARIANCE OF THE PROGRAM PARALLELISM.

3. THE INTRODUCTION OF ADDITIONAL PIPELINE BUFFERING SHOULD IMPROVE SPEED-UP CURVES CONSIDERABLY.

4. THE INTENDED CLOCK SPEEDS AND NUMBER OF PEs SHOULD GIVE A REASONABLE MATCH OF PROCESSING RATE TO PIPELINE BEAT FOR FLOATING POINT CODES.

5. MORE WORK NEEDED ON THE MATCHING STORE AND OVERFLOW UNIT.

6. NO REALISTIC ASSESSMENT OF THE VALUE OF A DATAFLOW MIP YET AVAILABLE.